REAL-TIME DATA MINING FOR PROCESS OPERATIONS USING GRAPHICS PROCESSING UNIT (GPU)-BASED HIGH PERFORMANCE COMPUTING

LAU MAI CHAN

NATIONAL UNIVERSITY OF SINGAPORE

2014

REAL-TIME DATA MINING FOR PROCESS OPERATIONS USING GRAPHICS PROCESSING UNIT (GPU)-BASED HIGH PERFORMANCE COMPUTING

LAU MAI CHAN

(B. TECH., NATIONAL UNIVERSITY OF SINGAPORE)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF CHEMICAL AND BIOMOLECULAR ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2014

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

_____

Lau Mai Chan

First submission on 8th August 2014

Revised on 4th March 2015

Acknowledgements

Foremost, I would like to express my very great appreciation to Professor Rajagopalan Srinivasan and Professor I. A. Karimi, my thesis supervisors for their patient guidance, continuous support and valuable advice throughout this research work. Without their persistent help this thesis would not have been possible.

Besides my supervisors, I would also like to offer my special thanks to my Thesis Advisory Committee, Prof. Lakshminarayanan Samavedham and Prof. Arthur Tay, for their useful critiques, insightful comments and encouragement.

I want to thank the departmental staffs, Mdm. Samantha Fam, Mr. Kok Hong Boey, Ms. Swee Yoke Woon, Ms Vanessa and Mr. Rajamohan, for being helpful and friendly. I take this opportunity to thank my fellow lab mates in Intelligent Applications in Chemical Engineering (iACE) group, Sathish Natrajan, Kaushik Ghosh, and Sha Meng for their generous help and support. Also I thank my friends at Institute of Chemical & Engineering Sciences (ICES), Arief Adhitya, Balaji Balgurunathan and Jonnalagadda Sudhakar.

Finally, I would like to express the deepest gratitude to my parents, my sisters and my brother for their unconditional support throughout my life. To my fiancé Daniel, who has always been very supportive and caring, I thank you for your patience and love.

Table of Contents

Summary

Data mining is a computational tool which is used to transform massive amount of process data into useful forms like models, patterns, or rules. The mined information can provide insights into the underlying process behavior, make (accurate) prediction about future process states, or detect anomalies, thus offering opportunities for enhanced process operations. In order to sustain optimal operational performance, the changing process behavior in real-world chemical processes needs to be captured accurately and efficiently. This thesis strives to explore the use of Graphics Processing Unit (GPU) parallel computing for developing computationally efficient data mining algorithms which are capable of handling large-scale, real-time chemical process problems. The emphasis of this thesis is on three important applications - continuous optimization, real-time image analysis and data stream clustering.

GPU is a computationally intensive device which consists of massive amount of processing cores. The processors are structured in a way that the same set of computing instructions can be executed on different data elements, in a parallel manner. Such data parallelization allows GPU to achieve significant enhancement in computational efficiency. As costly memory transfer between Central Processing Unit (CPU) and GPU is necessary for GPU implementation, it is important to ensure that there is sufficient amount of data parallelism for an overall satisfactory computational performance. On top of that, adequate understanding of GPU architectural and programming model, e.g. the size and access pattern of different GPU memories, are also important for better utilization of device resources and for developing highly efficient programs.

First, a systematic procedure for developing efficient GPU-based Simulation-Optimization (Sim-Opt) algorithms is proposed. By performing a comprehensive

structural analysis on Sim-Opt, we successfully reveal the various types of data parallelism for GPU acceleration. Based on these insights, a variable selection problem using Genetic Algorithm has been demonstrated on the proposed GPU-based procedure. In developing the GPU-based algorithm, we exploit data parallelism from independent operations among the candidate solutions in each GA generation, and also from the training and testing samples during PCA modeling and testing.

Next, a real-time multivariate image analysis solution is presented, wherein real-time particle size estimation is achieved by accelerating the algorithm on GPU. Besides using the pixel-level parallelization which is a common strategy of standard GPU-based image processing techniques, we also combine multiple feature extractors in single GPU function or kernel so as to avoid unnecessary memory access, and explore the use of GPU computing to accelerate multi-way principal component analysis (PCA).

Lastly, we propose a GPU-based online process monitoring scheme, in which a scalable clustering algorithm is developed to provide fine and accurate data summaries for effective model construction. The proposed scheme is particularly applicable to process monitoring problems which involve large amount of process variables and fast-arriving online process data. The massive computing power of GPU is exploited primarily by processing data in batches, and treating tree nodes at the same level concurrently. Besides, several GPU supporting features like global node pointers, memory pre-allocation, sorting, compacting, and scanning of tree nodes, are also proposed in this work. Memory pre-allocation of tree nodes provides a straightforward solution to the ineffective memory use in the original method which allocates tree nodes dynamically. Global node pointers which are associated to the pre-allocated tree nodes are a handy tool for nodes re-allocation, without actually moving their physical memory locations.

Sorting and scanning of tree nodes facilitate the identification of similar nodes, e.g. identifying nodes that are under the same parent node, which is important for parallel execution; while compacting is mainly used to segregate occupied pointers from those that are unoccupied.

The abovementioned developments have been tested extensively for accuracy as well as computational efficiency, using various case studies - the Tennessee Eastman challenge problem, batch crystallization operation, simulated oil and gas production operation.

List of Tables

List of Figures

Acronym Table

ACO	Ant Colony Optimization
ALUs	Arithmetic Logic Units
ANN	Artificial Neural Network
API	Application Program Interface
BIRCH	Balanced Iterative Reducing and Clustering using Hierarchies
CF	Clustering Feature
CLIQUE	CLustering In QUEst
CluTree	Cluster of Trees
CluStream	Stream Clustering
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CURE	Clustering Using REpresentatives
DBSCAN	Density-Based Spatial Clustering of Applications with Noise
DENCLUE	DENsity- based CLUstEring
DenStream	Density-Based Clustering over an Evolving Data Stream with Noise
EKF	Extended Kalman filter
EM	Expectation-Maximization
FDA	Fisher discriminant analysis
FDM	Frequency Domain Method
FFT	Fast Fourier Transform
FLOPS	FLoating-point Operations Per Second
GA	Genetic Algorithm
GMM	Gaussian Mixture Model
GP	Genetic Programming

GPGPU	General-Purpose computing on Graphics Processing Unit
GPU	Graphics Processing Unit
HAC	Hierarchical Agglomerative Clustering
HDDSTREAM	Density-based projected clustering over high dimensional data streams
HPC	High Performance Computing
HPStream	High-dimensional, Projected data Stream clustering
JC	Jaccard Coefficient
LR	Likelihood Ratio
KDD	Knowledge Discovery and Data Mining
KM	K-Means
MCL	Maximum Chord Length
MIA	Multivariate Image Analysis
MPC	Model Predictive Controller
NIPALS	Non-linear iterative partial least squares
NLP	Non-Linear Programming
NN	Neural Network
NSGA	Non-dominated Sorting Genetic Algorithm
ODAC	Online Divisive-Agglomerative Clustering
OO	Ordinal Optimization
QAP	Quadratic Assignment Problem
PA	Perturbation Analysis
PCA	Principal Component Analysis
PCI	Peripheral Component Interconnect
PLS	Partial Least Squares
PSVM	Proximal Support Vector Machine
PSO	Particle Swarm Optimization

PVM	Particle Vision and Measurement
RTO	Real Time Optimization
RSM	Response Surface Methodology
SA	Simulated Annealing
SFUs	Special Function Units
Sim-Opt	Simulation Optimization
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
SOM	Self-Organizing Map
SP	Streaming Processors
SPE	Squared Prediction Error
SPMD	Single Program Multiple Data
STING	STatistical INformation Grid
SVM	Support Vector Machine
TE	Tennessee Eastman
TSP	Travelling Salesman Problem

Chapter 1. Introduction

This chapter elucidates the motivation, research scope, methodology as well as the objectives of this thesis work. The challenges and limitations of existing techniques are also provided here.

1.1. Background and Motivation

Knowledge Discovery and Data Mining (KDD) is a computational process involving data selection, data cleaning, data transformation, data mining and information evaluation. Data mining is the most important step in KDD and its major role is to extract previously unknown information from large datasets, and then transform the discovered information into an understandable and actionable structure such as a pattern or model. The mined patterns or models provide an effective way for interpreting and analyzing the physical behavior of the associated process, due to their compact and precise structure. Better understanding of process behavior allows process engineers to devise effective planning, make effective purchasing decision, and optimize process for greater operational efficiency, better anticipate order lead times, or shorten machine downtimes (Sajdak, 2013; Rong, 2008; Wang, 1998; Psichogios & Ungar, 1992). Data mining models can also be used to predict process states or faults, which enable process operators or controller to anticipate and take appropriate actions accordingly (Seider, 2014; Yu, 2011; Doan & Srinivasan, 2005; Lee et al., 2004; Henson, 1998).

Real-time data mining is an advanced form of conventional data mining which incorporates additional features such as consideration of the age of the data, continuous model updating and handling of streaming data or big data (Chachuat et al., 2009; Lee et al., 2003; Saraiva, 1992; Zhang, 2002). Conventional data mining assumes invariant process behavior, so the models are built based on historical data which

resides in secondary storage device or data warehouse and is assumed to be representative of the process at all times. Thus, in the offline mode data mining is performed only once or infrequently. However, in real world chemical plants, even in the absence of faults, a normal process could undergo slow drifts, for instance as a result of, equipment degradation due to corrosion, catalyst deactivation that results in kinetic drift, wear-and-tear of parts, fouling and periodic maintenance or cleaning of machines. In this thesis, we focus on real-time data mining applications which are either used directly for process monitoring or employed for continuous updating of process models, and the ultimate objective is to ensure sustainable optimal process performance. As highlighted in the work (Bunin, 2014; Pahija, 2014; Yip & Marlin, 2004), adequate model complexity is the key for an accurate model which in turns guarantees a successful process optimization application. Similarly, in process monitoring applications, the outdated model might create false alarms which lead to confusion, unnecessary machine downtime, and wastage of manpower and cost (Cicciotti, 2014; Lee et al., 2003). It is generally true that a sufficiently large amount of process data is unavoidable for constructing complex and detailed models. Considering the high data generation speed in modern chemical plant, especially when high dimensionality is involved, traditional data mining algorithms are limited by their deficient computational speed and thus are not suitable in real-time applications.

The computational deficiency of many traditional data mining algorithms therefore offers the motivation for this PhD thesis, which endeavors to develop effective data mining algorithms for real-time extraction of process information. This thesis investigates key chemical process applications where the realization of real-time data mining is crucial.

1.2. Challenges of Real-Time Data Mining

Real-time data mining is challenging mainly due to the requirements of producing accurate information in a timely manner and handling huge volume of data or transient data with limited computational resources. As conventional data mining process is usually performed infrequently and offline, it focuses exclusively on accuracy and places limited interest on computational efficiency. On the other hand, an important feature of real-time data mining application is that the data mining algorithm is executed frequently at regular or irregular intervals. Hence, it is important that the algorithm is computationally efficient so as to prevent data backlog and also ensure timely information. Accurate and timely information is extremely valuable as it provides opportunity for continuously improving operational performance. However, due to the limitation of computational resources (i.e. processing power and memory), computational efficiency or accuracy is sometimes being compromised. Some existing works (Prata, 2009; Yip, 2004) are constrained to simple process like single-reactor polymerization process and a stand-alone boiler network; while some sacrifice accuracy to a certain extent for speed (Tosukhowong, 2004; Yue, 2004) by using simplified models or sampling techniques. Hence, the development of accurate and efficient data mining algorithms for real-time applications is highly attractive.

The large volume and high dimensionality of process measurement further complicates real-time data mining. As a result of advancements in measurement, sensor and network technologies, it is now a common practice in chemical plants to collect process data at extremely detailed levels which are characterized by high dimensionality and high data rate. Plant-wide or organization-wide data is gathered at a central storage place through data network if global-scale data analysis is of interest. To handle such a

massive amount of data, a data model called data stream is considered. A data stream is a finite or infinite sequence of ordered instances arriving at high speed. Due to the high arrival speed and the large volume, data stream algorithms are allowed to access data only once or a small number of times; this access mode is called a linear scan. Conventional data mining algorithms typically require random access to data. For instance k-means clustering repeatedly reads the data in random order. Such algorithms are therefore unsuitable for handling data streams. It is therefore evident that many offline data mining algorithms are not directly usable for real-time applications, and minor or major modifications may be needed.

1.3. GPU-HPC Solution. Why?

High performance computing (HPC) system is a powerful computing platform which has been successfully used for solving real-time problems. HPC solutions offer enhanced computational efficiency without sacrificing solution quality, based on a concept called parallelization. Standard HPC systems include supercomputers, grids, cloud, and computer clusters, wherein a large number of Central Processing Units (CPUs) are linked in different ways and work cooperatively for solving a big problem. In recent years, a new form of HPC system using General Purpose Graphics Processing Units (GPGPUs or GPUs for brevity) has emerged; a GPU the system consists of a single or a number of GPUs. Both CPU and GPU-based HPC systems serve as a parallel computing platform which provides many processing units for parallel works. If a computational problem can be divided into a number of smaller tasks which are independent of one another, the tasks can be executed simultaneously on a suitable HPC system, thus achieving enhanced computational efficiency. Successful applications report orders of magnitude acceleration achieved by parallel computing systems, for

example (Cano, 2014; You, 2014; Fok et al., 2007; Goil, 1997; Han et al., 2000; Liao, 2009; Liu et al., 2004; Melab et al., 2006).

CPU and GPU-based HPC systems differ in various aspects including the types of parallelization, costs and programming effort. On the one hand, CPU devotes majority of its transistors to control logics, so it has complete and complex control functionality for complicated computational tasks. Moreover, a CPU is designed to execute one process (or thread) at one time, at optimal efficiency. Hence, with multiple CPUs connected in a HPC system, it supports task-level parallelization, whereby each CPU executes the same or different task (i.e. program) on the same or different data. Due to the longstanding history of CPU in mainstream computers, majority of the commercial software or freeware are written and optimized for CPU implementation. There also exists a large body of well-established CPU libraries. As a result, less programming effort is needed when a single-CPU program is ported to a CPU-HPC system. However, the cost of building and maintenance of such CPU-based HPC systems is high which greatly reduces its feasibility for most applications.

GPUs on the other hand, have an entirely different architecture from CPUs and are targeted at a different type of parallelization. In a GPU, massive amount of transistors is allocated for arithmetic units, while only a few are used for control units. This GPU architecture allows for parallel execution of the same program (simple control) on different data (massive amount of arithmetic units). It thus enables a different type of parallelization, called data parallelization. GPGPU is a newly emerged technology which is still at its infancy. Owing to the different architecture, existing CPU programs are not directly executable in a GPU. In fact, significant amount of programming effort is required to develop an efficient GPU algorithm. Recently, a number of high-level

programming languages are being developed for GPGPU applications, such as Nvidia's CUDA, OpenACC and OpenCL, which simplify the programming. Besides the efficiency improvement offered by GPU, other attractive features of GPU include low cost, low power consumption as well as space saving. Therefore, we believe that GPU-based data mining algorithms offer a promising and practical solution to real-time information extraction in chemical plants. In this work, we explore the use of GPU computing to achieve real-time data mining for important chemical process applications.

1.4. Research Scope and Contributions

Figure 1-1 elucidates the research scope as well as the key objective of this thesis. In this work, we focus mainly on the data mining process which is known to be the most critical stage in KDD, and assume a minimum need on data pre-processing. As the data used in the case studies are either simulation data (Chapter 4 and Chapter 6) or reasonably complete data in the absent of missing values (Chapter 5), standard normalization should be sufficient. Furthermore, we are interested in tackling chemical process applications where real-time data mining is the key to assure the highest operational safety and optimal business value, specifically include the process control and fault detection problems. My major contributions include firstly to outset and explore the use of GPGPU parallel computing technology in realizing the practical use of data mining in chemical process applications, and this is accomplished by performing detailed examination on important data mining algorithms from which various types of data parallelism are successfully identified. Secondly, we demonstrate the development and implementation of several GPU-based data mining algorithms, and also address a number of important implementation issues related to GPU parallel computing. The developed GPU-based data mining methods have shown favorable results in terms of

computational speed and process control or monitoring quality, wherein it reveals the usefulness and potential of GPU technique in mining chemical process information.

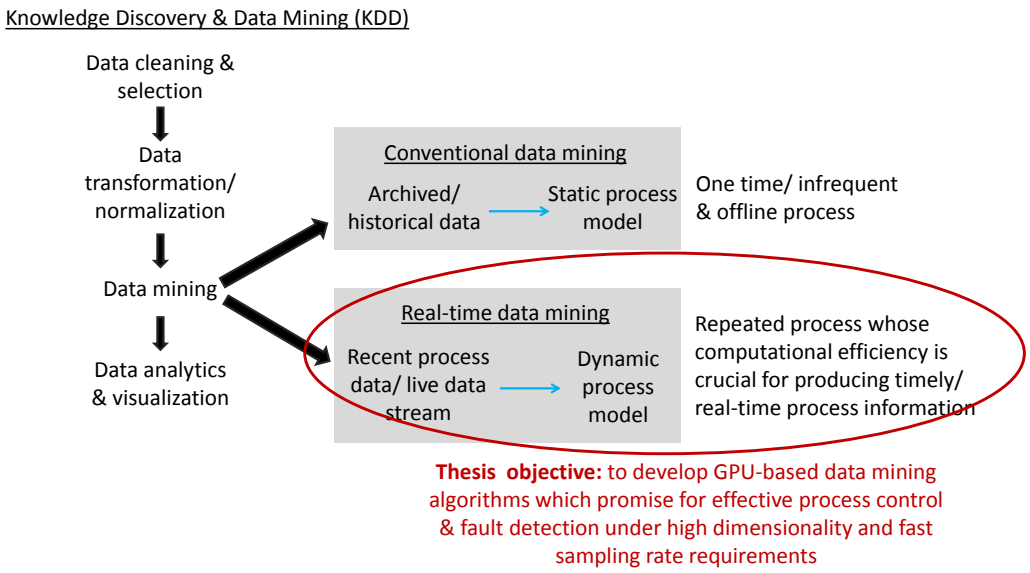

Figure 1-1-1: Research scope and objective of this thesis.

1.5. Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides a detailed review on the key concepts of data mining as well as various data mining applications in chemical industries. In this chapter, the state-of-the-art of real-time data mining technology is highlighted and existing research gaps identified. We focus specifically on 3 topics - continuous optimization, continuous pattern recognition using image analysis and clustering techniques. In Chapter 3, a comprehensive analysis of GPGPU parallel computing tool is performed. It focuses on the hardware architecture as well as software programming model of NVIDIA's CUDA-enabled GPU. Key programming issues and commonly used optimization techniques are also discussed. A survey of GPGPU-based research works is also presented.

In Chapter 4, we seek to improve the computational efficiency of simulation-optimization, a specific optimization technique, by devising a systematic procedure for

GPU programming. The implementation of the proposed procedure is demonstrated on a variable selection monitoring problem Enormous improvement in computational efficiency is obtained when the GPU-based algorithm is evaluated on the Tennessee Eastman Challenge Problem. Chapter 5 is devoted to real-time image analysis, where a GPU-based multivariate image analysis algorithm is developed. The proposed CPU-based algorithm has been tested successfully in a case study of particle size estimation for a batch crystallization process, for which particle size information is determined in real-time. Chapter 6 presents a GPU-based online process monitoring scheme. Essentially, a scalable clustering method known as BIRCH is ported to GPU so as to achieve an enhanced characterization of process behavior. Both synthetic and simulated process data are employed for evaluating the proposed approach. In the final chapter, key conclusions of the thesis are summarized and major research areas for future work are identified.

Chapter 2. Literature Review

This chapter begins with the introduction to KDD and data mining by including the definition and highlighting the key features of the technologies. It is followed by a literature review on various data mining methods and their classification. The importance of real-time data mining is then elaborated before detailed literature studies are performed on two important topics which are the continuous optimization and continuous pattern recognition.

2.1. Overview of Knowledge Discovery and Data Mining (KDD)

Knowledge Discovery and Data Mining (KDD) is an interactive and iterative computational process comprising of a number of phases including data selection, data cleaning, data transformation, data mining and information evaluation. This formal definition of KDD and the accompanying unified framework (as depicted in Figure 1-1) are firstly introduced by Fayyad and co-workers in (Fayyad et al., 1996; Fayyad et al., 1999). Such framework is widely accepted and has been continuously practiced by many researchers (Mariscal et al., 2010; Mhamdi & Elloumi, 2010; Sharma et al., 2012), though different abbreviations might be used such as the KDDM in (Sharma, 2012). Historically, knowledge discovery is realized through manual data analysis whereby human specialists are involved in understanding and deriving insights from raw data. As the amount of data collected across many fields continues to grow significantly, manual methods become impractical and inaccurate and thus computational data mining techniques emerge as a better, cheaper and less subjective alternative.

KDD is a multi-step framework wherein some or all of the component steps are automated. Each individual step plays a crucial role in determining the success of a

knowledge discovery application, and they are briefly presented as below. The first step, data selection, is used to remove irrelevant data records or variables which tend to pollute or dilute the important information. It is typically achieved based on certain prior knowledge either manually or by setting certain threshold value. The selected data subset goes through a data cleaning or preprocessing step whereby noise, errors and missing data fields are handled in a systematic manner. Fully automated computational data cleaning methods are widely available. It should be noted that the abovementioned data refinement steps can be accomplished through manual investigation if the volume of data is manageable small, as human is capable of differentiating good data from the bad ones. Subsequently, a more useful feature space could be obtained through dimensionality reduction, feature selection, variable transformation, or space projection, using the refined data; popular computational methods include PCA, PLS, and SOM. In this third step, invariant representations for the original data can be produced from the transformed feature space, and thus creating a cleaner data space for analysis.

The remaining steps of the KDD framework are data mining and information evaluation. The fourth step, data mining, is considered as the most computationally intensive and complex step in KDD. This could explain why it is the main focus of this work. The key objective of data mining is to search for interesting and useful patterns which can be in the forms of models, decision tree, classification or association rules, or cluster assignment; a more comprehensive literature review will be provided in following sections. As there are a wide range of data mining techniques, from simple regression methods to machine learning and artificial intelligence techniques, choosing a suitable method becomes a nontrivial task and caution has to be taken. The following step, information evaluation, has therefore become increasingly important as the

volume of data mining techniques keeps growing. The evaluation step which is the final step in the KDD cycle involves thorough review on all the preceding steps and careful examination on the data mining results. At this phase, it depends heavily on human involvement to cautiously interpret the discovered patterns, and it is done based on expert knowledge and with the help of visualization methods if possible. The objective is to make judgment on the use of data mining results by removing irrelevant or redundant patterns, and translate the useful results into forms which are comprehensible and implementable by users. If the extracted patterns are not useful or ambiguous, the KDD cycle needs to be repeated and alternative data mining techniques will be considered. Although the evaluation step is not covered in details in this work, it plays an equally important role as all the preceding steps in KDD.

2.2. Introduction to Data Mining

Data mining involves the extraction of hidden, previously unknown interesting and useful information from large volume of data, wherein the discovered information is in the form of understandable or actionable structures such as rules or models (Fayyad et al., 1996; Mukhopadhyay et al., 2014; Mahmood et al., 2013). For discovery of useful information, large number of measurements and many attributes (or variables) need to be simultaneously explored, thus manual analysis is impractical. Data mining techniques which include a wide variety of machine learning algorithms offer an alternative feasible solution. The design of a data mining method requires paying attention to the following issues (Fayyad et al., 1999).

Model representation: In the context of data mining, model is a concise representation of the discovered information or pattern. It consists of two parts which are the model structure (e.g. linear, nonlinear, Gaussian model, PCA) and model parameter (e.g. mean

and standard deviation of Gaussian model, eigenvectors of PCA model). Model selection depends heavily on the prior knowledge of the domain or system as well as the application goal. Compact and neat mathematical models are sometimes inadequate in describing highly complex systems (Quaglia et al., 2015; Yip & Marlin, 2004). However, excessive model complexity, like in the case where there are too many layers employed in an artificial neural network (ANN), has the tendency of over-fitting which leads to poor generalization. The rule of thumb is to choose the simpler model if its performance is adequate, according to Occam's razor theory.

Search method or learning algorithm: A good search method is one that provides effectiveness and efficiency in attaining the fittest model. There are two levels of searching where the top level is to identify a suitable model structure (e.g. the order of polynomial), and the bottom level is to find the optimal set of parameters for the selected model structure. Search method can be broadly divided into two groups, global search and local search. Global search methods like genetic algorithm (GA) and swarm-based optimization explore a wider solution space, while local search methods like hill climbing and k-means look at solutions which are usually at near proximity. The former is generally more computationally demanding but offers higher chance of getting the best model, whereas the latter requires less computing effort but it only guarantees for suboptimal solution.

Model evaluation: A quantitative measurement of the model fitness (i.e. fitness function) is required to ensure the model performs satisfactorily and meets the data mining goal. In applications where the fitted model is used for prediction, classification error and prediction accuracy (e.g. mean squared error) are common choices. When the data mining problem involves characterizing a system or process based on clustering

algorithms, similarity of items in the same cluster should be maximized while dissimilarity between items in different clusters should be minimized. In general, a separate set of data (i.e. testing data) is needed for model evaluation, this is to avoid over-fitting and promote model generalization which guarantees good prediction for unseen data.

2.3. Classification of Data Mining Methods

Numerous data mining algorithms have been developed with each mines for different type of information, one scheme of classifying them (Clifton, 2014) is based on the type of known information and the kind of information to be searched for. It should be noted that the same method can exist with different interpretation to suit the goal of application, and therefore it may fall into more than one category.

Predictive modeling: This group of algorithms provides estimation of class or numeric value of the target variable or output. Classifiers, like support vector machine (SVM), neural network (NN) and naive Bayes classifiers, can serve as a predictive model with categorical target variable. While in regression methods, such as linear regression, non-linear regression and neural network, the target variable is numerical or real-valued. For instance, (Chiang et al., 2004) examines the proficiencies of Fisher discriminant analysis (FDA), SVM, and proximal support vector machines (PSVM) in classifying multiple fault classes; (Gonzaga et al., 2009) proposes a feed-forward artificial neural network (ANN) to estimate PET viscosity online. The powerful approximating capability of neural network allows it to become a popular candidate for process control (Assenhaimer et al., 2014; Singh et al., 2014; Bhat, 1990; Nahas, 1992), by providing accurate prediction of process output.

Descriptive modeling: The main objective is data characterization, which is achieved by extracting patterns through grouping data items according to certain similarity measure; clustering is one of the most common methods. Similarity is measured based on distance metric e.g. Euclidean distance. A key difference between descriptive modeling and predictive modeling is that the former is usually trained using unsupervised technique, while the latter requires supervised training. Hence, clustering-based pattern recognition is particularly useful in cases where there is no prior knowledge (no labeling) about the data, e.g. (Detroja et al., 2006) uses clustering to detect faults that have not seen during training; (Srinivasan et al., 2004) proposes a two-step clustering method for automatic identification of different process states based on historical data. Clustering is a well-established data mining algorithms and it can be broadly categorized into hierarchical clustering (e.g. HAC, BIRCH, and CURE), partitioning clustering (e.g. K-means and K-medoids), density-based clustering (e.g. DBSCAN and DENCLUE) and grid-based clustering (e.g. STING and CLIQUE) (Pham & Afify, 2007). A more detailed investigation of clustering techniques will be provided in Section 2.6.2.

Pattern mining: In this category, the extracted pattern is in the form of rules; association rules and sequential rules belong to this category. It focuses on discovering interesting relations between variables in large databases. In (Ren et al., 2005), an association rules mining technique is proposed and applied for reasoning about the relationship between recovery rate and energy loss in a real aromatic hydrocarbon extraction process. Zhang et al. (2005) use association rule mining to establish the relationships between the key process variables and some objective variables, like purity and recovery rate, of a simulated moving bed paraxylene adsorption process.

Anomaly detection: The major goal is to detect anomaly by looking for data items that are usual and do not fit the established models of normal process data. While predictive modeling can be used for detecting known or trained faults, there could be novel and unexpected faults appear in the process, and their discovery is highly valuable. A popular method in this category is principal component analysis (PCA) (Choi et al., 2004; Ku, 1995; Nomikos, 1994). Here, the measurement of Hotelling's T^2 statistic and squared prediction error (SPE) on the PCA model provide useful indication of process abnormality.

2.4. Real-Time Data Mining and Its Desirable Characteristics

Real-time data mining can be regarded as a continuous data mining process to cope with the changing environment conditions by continuously searching for new information and making adaptation to model. It enables continuous monitoring of chemical process operation so that quick responses and real-time decisions can be made in response to supply disturbances, market uncertainties, process behavior changes, process abnormalities, equipment malfunctions and faulty sensors. With effective real-time data mining system in place, it is possible to maintain the process operational performance, product quality, process safety and customer's satisfaction at the optimal level.

In real-time data mining applications, it is exceptionally important to ensure that information is obtained in a promptly manner. Conventional data mining algorithms which are computationally expensive and bounded to disk-resident data are therefore not suitable. They need to be revised and improved to address the following issues (Fan,

2013; Gaber, 2005; Gama, 2012; Silva, 2013) before they can be used in solving real-time data mining problems.

Unbounded and rapid arriving of data: In modern chemical plants, owing the rise of information technology and advances in sensor technology, the volume of data grows at an unprecedented rate. Various types of data which may be related to processes, products, scheduling, planning, equipment, maintenance and inventories are generated continuously and infinitely. Most of the time, data generation and collection rate is much faster than data mining speed. The capability of closing this time gap is thus an essential feature of real-time data mining techniques.

Modeling process changes over time: The behavior of real world chemical processes is never stationary, abrupt changes like equipment malfunction, actuator failure and slow changes like equipment degradation, catalyst deactivation, are taking place consistently. In response to abrupt changes, an effective fault detection and remedial action is possible only if the faulty condition is captured accurately and quickly without delay. Although slow changes or drifts may not result in sudden and serious consequences, failing to detect such drifts would lead to suboptimal or deteriorating operational performance. Outdated model in control application could drive the process to suboptimal or negative performance direction; in process monitoring, it could result in fault positive (i.e. false alarm) or fault negative detection. The challenge is how to detect process changes over time and enable adaptive modeling.

Constraints of computing resources: Most data mining algorithms require data to reside in main memory, so in cases where data size is larger than available memory size, it is required to move data in and out from secondary storage device which is extremely inefficient. Furthermore, high arrival rate and huge volume of data prohibit its long

residence in main memory i.e. data is transient. Transient data is disallowed to change its order due to time and space limitation. The challenge is to design a data mining method that is space-efficient and guarantees for useful pattern extraction under the constraint of linear scan to data.

In this thesis, our focus is on devising feasible and effective real-time data mining solutions to three important chemical process applications, they are continuous optimization, and continuous pattern recognition using image analysis and clustering techniques. It should be noted that we do not address the issues regarding early stages of KDD, including data collection, data cleaning, data preprocessing, in this work. Interested readers are referred to (Fayyad et al., 1996, 1999; Li, 2009; Mariscal, 2010) for details.

2.5. Continuous Optimization

Optimization algorithms provide a means to search for the best solution while satisfying certain constraints. In the context of chemical process operations, a best solution could be a set of adjustable process variables corresponds to the highest possible production rate while ensuring pre-specified product quality (i.e. constraint), or an optimal allocation of limited resources and assets to satisfy market demand, or it could be simply an accurate prediction model for fault detection. Common optimization applications in chemical industries include process operation optimization (e.g. supply chain management, process planning and scheduling) (Gupta, 2003; Méndez, 2006), process model building (Yu, 2008; Psichogios & Ungar, 1992), process design and synthesis (e.g. reactors network configuration, heat exchanger) (Karuppiah, 2006; Papoulias, 1983; Yee, 1990) and process control (Biegler et al., 2002). Data mining strategies such as association, classification, and clustering are always useful in

optimization applications, through providing concise and accurate process model. Take, for example (Yu & Qin, 2008) uses Gaussian mixture model (GMM) to capture the multiple operating modes of complex industrial processes, for the purpose of process monitoring. In the study, the GMM model is established by determining the model parameters through optimization. In (Psichogios & Ungar, 1992), the authors propose a hybrid modeling process by combining the use of data-driven neural network and first principles. Through machine learning optimization, the optimal structure and parameters of ANN are identified for accurate process variable estimation of a fed-batch bioreactor. The above examples show the use of optimization in process model building.

Continuous optimization is an increasingly popular research topic in recent years due to the capability of maintaining the optimal performance at all times. In real world chemical plants, the ever-changing behavior of processes causes a previously optimal operation condition or design no longer the best performing one in later time. In order to cope with such changing behavior, it is required to perform the optimization routine repeatedly. (Biegler & Zavala, 2009) illustrates how a real-time optimization (RTO) can be used in combination with a model predictive controller (MPC) in achieving smooth and optimal transitions between different operating modes. (Tosukhowong et al., 2004) is another research work which integrates the use of RTO and MPC, but the authors suggest using an intermediate frequency for optimization. The proposed approach guarantees for lower economic loss as compared to a less frequent RTO (i.e. stead-state RTO), in the presence of disturbance. In these two studies, the role of RTO is mainly to identify the best performing set-point for MPC. In the study (Prata et al., 2009), the authors employ particle swarm optimization method (PSO) to iteratively update data model parameters. The proposed procedure is tested successfully in the operation of a propylene polymerization process, by having accurate and real-time

prediction of process variables. Such continuously adapted process model is also demonstrated to attain a better stabilization of control loops at plant site.

From the perspective of data mining, continuous optimization involves continuous model adaptation or re-construction based on most recent data. The relationship between optimization and data mining can be two-fold, where recurring optimization is either used as a model updating tool or is performed based on the updated model. Some examples are detailed below. (Peters et al., 2007) propose a real-time optimization approach aiming to continuously improve the overall cost of batch processes, based on a regularly updated process model. However, the searching method is limited to a smooth and locally convex cost function. The research work (Golshan et al., 2005) suggests using the Extended Kalman filter (EKF) to estimate process states and time-varying parameters of the process model. The updated process model is then optimized through NLP to determine an optimal set of process operating conditions at every 8 hours. EKF does not guarantee for the optimal estimation of process behavior as a consequence of the linearization requirement when it extends from the regular Kalman filter.

To overcome the limitations of conventional optimization methods e.g. EKF and NLP, the authors of (Prata et al., 2009) propose the use of heuristic optimization method PSO which is capable of handling high dimensional problems, attaining globally optimal solutions and allowing for straightforward implementations. In the study, model parameters and process states are estimated by performing PSO on moving windows. Moving window or moving horizon is generally used to reduce the size of optimization problems and at the same time discard old process data. A survey of various model adaptation strategies for real-time optimization applications is presented in (Chachuat

et al., 2009). As it has been pointed out in the article, besides model parameter adaptation, there are also situations where we should take in consideration of modifying the model structure, so as to avoid plant-model mismatch.

2.5.1. **Simulation-optimization**

Simulation-optimization (Sim-Opt) is a special type of optimization techniques which makes use of simulation to replace deterministic mathematical formulations. It provides plausible optimization solutions for systems involving complex phenomena and containing uncertainty, which are unsolvable otherwise. The concise mathematical expression for Sim-Opt is given as,

$$\min_{\theta \in \Theta} E\left[L\left(x(\theta, \xi)\right)\right] \qquad (2\text{-}1)$$

where, *L* is a performance measure, θ denotes variables or parameters subject to optimization, Θ refers to the entire search space for θ, ξ represents the system randomness, *x* is to the trajectory for θ generated based on ξ. Essentially, Sim-Opt comprises of three functional modules, which are simulation, objective evaluation and optimization. The information flow and relationship between the three modules are depicted in Figure 2-1. In Sim-Opt, the system reaction of all candidate solutions θs generated from an optimization cycle is first simulated, after which objective evaluation *L* is performed based on the simulated response $x(\theta, \xi)$. For a stochastic system where uncertainty is involved, multiple simulations with different paths ξ are needed for each candidate solution.

In modern chemical plants, highly integrated process configuration gives rise to complex interaction among process variables. Modeling of such processes requires large-scale and difficult mathematical formulations. Furthermore, uncertainty exists in

chemical processes in various forms, from raw material quality variation to fluctuation of market demand, so deterministic mathematical models are sometimes inadequate. Considering the capability of simulation in capturing system complexity and stochasticity, Sim-Opt techniques are highly favorable in solving complex optimization problems.

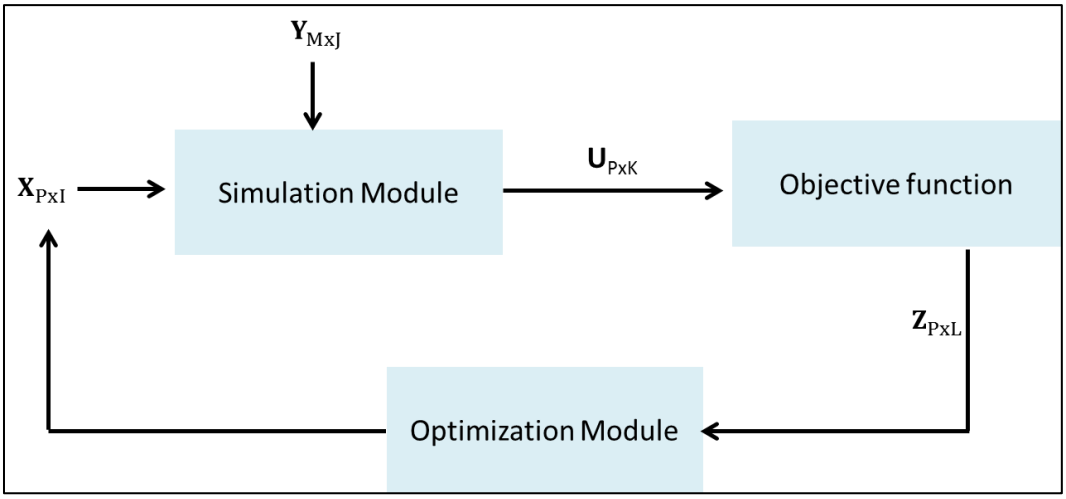

Figure 2-1: General structure of simulation-optimization

2.5.1.1. **Sim-Opt in chemical process operations**

Sim-Opt have attracted considerable attention from researchers and industrial practitioners in recent years. Reviews of Sim-Opt techniques are provided in (Wang & Shi, 2013; Azadivar, 1999; Swisher et al., 2000). Successful applications of Sim-Opt are found in areas such as process design and operation, supply chain management, and sustainability development. Brunet et al. (2012) combine simulation and optimization techniques to solve a single-product process design problem, in which both structural (type and size of process units, number of equipment units in parallel) and operating decisions (e.g. concentration, flow rates, temperature etc.) are taken care of. Reyes-Labarta et al. (2012) solve the design problem of multi-component distillation columns using Sim-Opt technique. This is a complex optimization problem where design variables

(including total number of stages and feed location) interact heavily with operating decisions (i.e. distillate flow rate and reflux ratio). Brunet et al. (2012) present a Sim-Opt methodology for designing absorption system which is used for cooling and refrigeration purpose. The design problem is solved by taking into consideration both economic and environmental factors. In the area of sustainability, Halim et al. (2011) propose a knowledge-based Sim-Opt framework for generating sustainable design and operations alternatives for chemical process plants. The above works demonstrate Sim-Opt applications that are motivated mainly by the capability of simulation on modeling complex systems, e.g. reaction kinetic, thermodynamic relation, vapor-liquid equilibrium, while there is limited or no uncertainty involved.

The uncertainty handling capability of simulation has been stressed in Sim-Opt applications found in the area of supply chain management. For example, Chen et al. (2012) present a Sim-Opt approach to minimize clinical trial costs during new drug development in pharmaceutical industry, whereby stochastic factors like patient enrollment rate, randomly assignment probability and dropout rate are taken into consideration. Mele et al. (2006) present a simulation-based optimization framework for optimal chemical supply chain management under demand uncertainty. Nikolopoulou et al. (2012) design a Sim-Opt approach to address the planning and scheduling decision problem in supply chain management, with demand uncertainty. Tan et al. (2010) address the business decision support problem in a global specialty chemicals enterprise using a Sim-Opt framework, in which decision variables such as reactor size and policy decision like raw material reorder point are optimized. The behavior, interaction and various uncertainties of supply chain entities are simulated using a dynamic model.

2.5.1.2. **Limitations of Sim-Opt**

Although Sim-Opt have been widely accepted as an effective optimization technique, the high computational cost associated with simulation has limited the practicality of the techniques. Assuming Sim-Opt are used in continuous optimization applications, the computation cost will become extremely huge and unmanageable with standard computing resources. As a result, real-time performance is nearly impossible. Detailed structural analysis of Sim-Opt will be given in 4.1, in which the effect of optimization technique and degree of uncertainty on computation time will be examined.

2.5.1.3. **Approaches for improving computational efficiency**

A number of studies attempt to overcome the high simulation cost in simulation-optimization applications, which can be broadly classified into two groups. One group of studies seeks for alternative optimization techniques that require lesser simulation runs, while the other group concentrates on enhancing computational speed based on parallel computing technology. The first group essentially refers to the special gradient estimators which have been mentioned early; classic examples include perturbation analysis (PA), likelihood ratios (LR), response surface methodology (RSM) and ordinal optimization (OO). A major drawback of the PA technique is that the estimated gradients are often biased and inconsistent (Carson & Maria, 1997). In addition, the technique is based on the assumption that perturbation made in an input variable does not affect the sequence of events, which may not be always true. Although LR guarantees for unbiased gradient, the method is not applicable to optimization problems that involve in-differentiable objective function (Fu, 1994). In RSM, regression polynomials or neural networks are used to approximate the relationship between objective variables and decision variables, based on simulation

data. Although the overall number of simulations is reduced, the method is restricted to smooth objective-to-decision variable relation, where there is no sharp ridges and flat valleys (Azadivar, 1999). On the other hand, OO focuses on finding good enough solutions (not the best solution) by exploring as much as the solution space, within certain computational time; where the global searching capability is enhanced at the cost of its local searching capability (Zhong et al., 2006). Since a crude model is firstly used for selecting a subset of good solutions, and simulation is performed only on the selected subset, computational time is significantly reduced. However, statistically good solutions can be difficult to obtain (Swisher et al., 2000; Zhong et al., 2006). While the aforementioned optimization approaches promise reduced number of time-consuming simulation runs, they are only applicable to over-simplified optimization cases.

The advent of parallel computing technology offers an alternative way of improving the computational efficiency of simulation-optimization techniques. Fujimoto (1989, 1990) and Misra (1986) discuss and compare various parallel computing techniques for accelerating single simulation, based on distributed computing systems like supercomputers and a network of processors. There are also studies, such as (Koo et al., 2008; Laganá et al., 2006; Tan et al., 2010), which work on multiple simulations at any one time. However, these algorithms require CPU-based HPC systems which can be extremely expensive.

2.6. Continuous Pattern Recognition

Pattern recognition can be regarded as a classification technique, in which machines attempt to learn from the environment and make judgment about the categories of the data observations or patterns. As other data mining techniques, the product of the learning process is a model. There are two broad groups of classification

techniques, namely supervised and unsupervised classification. In cases where the actual classes of training data are known, supervised classification techniques are usually more adequate; otherwise unsupervised methods are the only choice. The number of studies of applying pattern recognition techniques in chemical process operations is tremendous and growing continuously, some examples include process monitoring and fault diagnosis using SVM (Chiang, 2004; Zhang, 2009), PCA (Bakshi, 1998; Kresta, 1991; Lee et al. 2004) , partial least squares (PLS) (MacGregor et al., 1994), faulty sensor detection (Dunia et al., 1996; Qin & Li, 2001), process states identification (Srinivasan et al., 2004), and etc.

As business has become increasingly competitive and regulations are getting more stringent, continuous pattern recognition approaches which provide updated information are highly attractive. In reality, process behavior is dynamic and time-varying due to many reasons, e.g. seasonal fluctuation, tools aging, external disturbances, equipment degradation. Therefore, it is crucial to detect or capture process changes, so that adequate actions can be taken for maintaining optimal operational conditions or satisfying regulatory standards. Relevant studies include (Geng & Zhu, 2005) which presents an adaptive multiscale-nonlinear PCA algorithm for on-line monitoring of slow process changes. The adaptive feature is granted by the use of moving windows which acquire the process data dynamically. Instead of using moving windows, Yue et al. (2004) propose to assign different weights to samples according to their age. In doing so, old information will be forgotten gradually and newly constructed PCA model reflects the current process behavior. In (Lee et al., 2003), the authors demonstrate that the use of fixed multiway PCA model can lead to false alarms when it is applied in a real process monitoring problem; the issue is resolved with consecutively updating of the multiway PCA model. In (Ge et al., 2009), a statistical local method is

used to formulate changes of parameters in the kernel PCA model, so as to account for the changes of process behavior.

However, the existing studies are mainly restricted to the use of PCA-based strategy, and they show zero or little concern on computational efficiency. As model is continuously updated or re-constructed, it can severely slow down the pattern (fault or changes) identification process. Hence, our main goal is to develop efficient pattern recognition methods for 2 important types of algorithm, which are image analysis and clustering. In the following paragraphs, we discuss the state-of-the-art of these two groups of techniques as well as their applications.

2.6.1. **Image analysis**

Image analysis involves identification of the position, size, or shape of objects in a system, through classifying image pixels into separate groups representing background, edges, boundary or objects. A standard digital image can be expressed as a 2D matrix (gray-scale) or a three-way 2D matrix (RGB or color). The size of the 2D matrix indicates the degree of resolution, e.g. 720x480, 480x640, etc. On the other hand, spectral imaging techniques provide much richer details of a scene by integrating both the spatial and spectral information, in the form of 3D matrix, or a stack of 2D matrices. The 3 important stages in a standard image analysis process are discussed as follows (M. P. Ekstrom, 1984; Pitas, 1993).

Image enhancement: Image filters are the basic tool for image enhancement. It can be broadly categorized into two groups, which are the Fast Fourier Transform (frequency domain) and convolution (spatial domain) filtering. In this work, we focus on the latter. Convolution filtering performs discrete convolution of the original image with a special mask; where the fundamental idea is to assign each pixel a value depending on the

values of its neighboring pixels. The simplest mathematical formula of a convolution filter is given below,

$$g(x,y) = \sum_{i=-v}^{v} \sum_{j=-u}^{u} h(i,j) * f(x+i, y+j) \qquad (2\text{-}2)$$

where, *h* is a mask given as a matrix of size $(2v+1)\text{by}(2u+1)$, and the function or values of *h* is known as the filter kernel. Depending on the functionality and complexity of *h*, filters can be divided into low-pass filters and high-pass filters. Low-pass filters such as mean and median filtering are commonly applied for image de-noising or smoothing. Whereas high-pass filters e.g. Laplace, Roberts, Sobel are mainly used for image sharpening. As high-pass filters develop useful features such as edges, lines from raw image, they are also known as feature extractors.

Image segmentation: The objective of image segmentation is to partition an image into multiple segments of connecting pixels; a segment is a group of pixels with similar visual characteristics. The pixel segments which can be background, lines, curves, boundaries and etc. are used for locating and characterizing objects in the subsequent stage. The most popular method used for image segmentation is thresholding. Image thresholding process replaces the values of background pixels to 1's (i.e. white) and values of object pixels to 0's (i.e. black), thus the product is a binary image. There are several ways of obtaining the threshold value, some based on blank images while others rely on the considered image.

Image post-segmentation and characterization: Commonly, the objects identified during image segmentation are defective, thus further enhancements are required. Morphological operations like erosion, dilation, opening, closing are standard methods used at this stage, for filling missing holes, removing objects that are tiny and touching

the image boundary. For effective characterization of the objects, object boundaries need to be first identified by locating the connected pixels which separate the object from the background. Lastly, quantitative or qualitative characterization of the image can be performed.

2.6.1.1. **Applications in process operations**

Image analysis represents a powerful technique for monitoring and controlling the shape and size distribution of crystal products in pharmaceutical, specialty chemical, agrochemical industries. Numerous studies have been conducted, for examples, (De Calderon Anda et al., 2005; Eggers et al., 2008; Larsen et al., 2006, 2007; Oullion et al., 2007; Sarkar et al., 2009); some provide 3D size information (Darakis et al., 2010; Kempkes et al., 2010); some uses image analysis to support model development (Monnier et al., 1997).

On top of that, applications of image analysis are also found in other non-particulate processes. Yu et al. (2004) develop a multivariate image analysis technique for online performance monitoring of boiler system, by analyzing RGB flame images. By combining the information extracted from flame images with additional process data, it allows prediction of more useful information like concentration of NO_x and SO_2 in the off-gas. In (Khalil et al., 2010), an automatic image analysis technique is used for studying the effect of process parameters, e.g. surfactant concentration, stirring power, on the droplet size distribution in an emulsification process. The optimal process settings attained from the study show 3 to 4 times faster in reaching equilibrium. In the study (2009), the authors present a multi-resolution and multivariate image analysis method to identify defects on the surface of a photolithographed semiconductor device.

The use of image analysis is also extended to the study of gas-solid fluidization and mixing systems, where standard analytical measurement is impractical or ineffective. The features of bubbles, including size, moving speed, size evolution, have great impact on the fluidized process performance, so visualization and understanding of the bubble behavior are of foremost importance. Image analysis approaches are well-suited to this case, as shown in the works (Busciglio et al., 2009; Kantzas & Kalogerakis, 1996; Lim et al., 2007). While some studies of fluidized system focus on the particle flow dynamics like (Liu et al., 2008; Pallarès & Johnsson, 2006). On the other hand, the concentration of components during solid-solid powder mixing or liquid-liquid mixing which is difficult to measure can also be determined with image analysis techniques, as illustrated in (Blood et al., 2004; Li & Wei, 1999; Santomaso et al, 2004).

From the perspective of continuous (i.e. online) process monitoring and control, image analysis techniques which are capable of providing timely information about the process are extremely valuable. Although, in (Larsen et al., 2006, 2007), it is reported that the image analysis algorithm is sufficiently fast for real-time implementation, the image acquisition speed of the video camera system is relatively slow which is 12min per image. Many modern imaging systems operate at a much faster speed, for instance (Sarkar et al., 2009) employ a Particle Vision and Measurement probe captures image at every 0.5 second, (Yu & MacGregor, 2004) is based on imaging time of 1 frame per second, (Khalil et al., 2010) records video sequence at the speed of 30 seconds per image. In some cases, the use of image analysis is restricted to off-line applications, e.g. (Oullion et al., 2007).

2.6.2. **Data clustering**

Clustering is an unsupervised classification technique which divide non-trivial amount of objects into a smaller number of groups (i.e. clusters), in which objects in the same group share similar attributes and those in different groups are more dissimilar. Standard clustering methods include k-means, density-based clustering, hierarchical methods, fuzzy clustering, and neural network-based and genetic algorithm-based clustering, an overview of these methods can be found at (Pham & Afify, 2007). More detailed discussions on several key aspects of clustering e.g. object types, similarity and distance measure, clustering optimality are provided in (Grabmeier & Rudolph, 2002). These standard clustering algorithms are among the most popular data mining tools which have been successfully applied in many disciplines such as bio-informatics (Alon, et al. 1999; Edgar, 2010; Eisen et al., 1998; Li & Godzik, 2006), environmental science (Chen et al., 2007; Malmberg & Maskell, 2002), economics (Colin Cameron et al., 2011; Breschi & Malerba, 2001), and also chemical process operations as discussed below.

2.6.2.1. **Applications of conventional clustering algorithms in process operations**

Clustering emerges as an effective tool for process states identification, for example in the work (Srinivasan et al., 2004), a PCA-based two-step clustering method is proposed for classifying historical data into various process modes and transitions, which are characterized by different optimal control configurations. Ng et al. (2008) demonstrate the use of self-organizing map (SOM) for process modes identification; the approach has later been validated on process monitoring and diagnosis applications in (Ng Y. S., 2008). Wang et al. (2013) propose using a subtractive clustering algorithm for identifying the operation mode of a coke oven system, based on historical data. Correctly identified operation mode is important as it allows determining of optimal

pressure set-point for coke oven operation. A similar application is also presented in (Pyun, 2011) where k-means clustering is used to classify LNG data into several operating modes for process monitoring purpose.

In addition, clustering of chemical components into smaller and more manageable groups helps to reveal the chemical similarity and thus facilitating comparison, understanding and making references on the components. In (Aji et al., 2004), the PLS model constructed based on pre-clustered mid-distillates offers more accurate concentration prediction of aromatic components. Akman et al. (2008) propose a hierarchical clustering strategy to classify essential oil components in supercritical CO_2 phase. Clustering of components provides a clearer view of the relative distribution and enables optimal design and operation of supercritical fluid technology. Babaei Pourkargar et al. (2010) demonstrate the use of fuzzy clustering to predict thermodynamic and transport properties of hydrocarbons based on the cluster membership of the components. In some cases, plant facilities can also be clustered into smaller groups for more effective network design and operation optimization, e.g. design of biomass supply network using clustering technique (Ng & Lam, 2013), coherent power generator groups are identified using support vector clustering (Agrawal & Thukaram, 2013).

Direct applications of clustering in process fault detection and diagnosis have also been found in the literature, e.g. (Detroja et al., 2006). In the study, historical patterns are first classified into various clusters of normal or faulty operations using possibilistic clustering. During online monitoring phase, the cluster membership of plant data serves as the indicator of whether the data falls into normal or abnormal process conditions. Similar to the works discussed early, the clustering is performed off-line on a

rather small historical dataset, by assuming unchanged process behavior. Hence, conventional clustering methods are unsuitable for problems involving continuous pattern recognition.

2.6.2.2. **Data stream clustering**

Effective continuous clustering applications involve handling large volume of data or fast arriving data stream in an efficient manner. Apart from using expensive CPU-based HPC computing systems to induce scalability and efficiency, like in the studies (Dhillon & Modha, 2000; Feng et al., 2007; Kaur et al., 2012; Li et al., 2012; Li & Xi, 2011; Patwary et al., 2012; Tsui et al., 2012; Xu & Zhao, 2012; Yang et al., 2012), we are more interested in clustering methods which are fundamentally scalable to large data as they represent more practical alternatives. Review of these scalable methods can be found in (Silva et al., 2013; Yogita & Toshniwal, 2013).

Balanced iterative reducing and clustering using hierarchies (BIRCH) is one of the earliest and most widely used scalable clustering method proposed by Zhang et al. (1996). The key features contributing to the scalability of BIRCH are clustering feature (CF) and micro-clusters; while the tree structure organization of micro-clusters allows for efficient searching. The fundamental idea is to incrementally assign data to the most similar micro-cluster. The main characteristics and data distribution of a micro-cluster is represented by a tuple CF=(*N*,*LS*,*SS*), where *N* is the number of data, *LS* is the linear sum of the *N* data, *SS* is the squared sum of the *N* data. Through condensing *N* data points into a CF tuple and limiting the number of micro-clusters to be formed, BIRCH is able to take in infinite number of data theoretically. The resultant micro-clusters which represent data summaries can be further processed or clustered to generate more meaningful results. More details of BIRCH algorithm are provided in Section 2.6.2.3.

A number of scalable clustering algorithms have been derived from BIRCH, based on the CF-vector and micro-cluster concept. The CluTree algorithm proposed by Kranen et al. (2008) incorporates temporal information to tree nodes (i.e. micro-clusters), whereby CF computation is time dependent. In doing so, outdated information will slowly be forgotten. ODAC is another variant of BIRCH employing the micro-clustering approach, but it is a method for variables or features clustering instead of data clustering (Rodrigues, 2008). Some of the micro-clustering algorithms remove the tree structure; popular ones include DenStream, HPStream, HDDSTREAM and CluStream. The DenStream algorithm proposed in (Cao et al., 2006) constructs density-based micro-clusters which undertake arbitrary shape. It also employs a different pruning strategy to limit the amount of micro-clusters. Aggarwal et al. (2005) and Ntoutsi et al. (2012) develop micro-cluster-based HPStream and HDDSTREAM algorithms, respectively, which are targeted at high dimensional clustering problems. The CluStream algorithm removes outdated data by tracking and including the temporal information of micro-cluster into CF (Aggarwal et al., 2003).

While the abovementioned BIRCH-derived methods process data incrementally, i.e. one at a time, there are also algorithms working on data batches like the scaleKM algorithm proposed in (Bradley et al., 1998) and its enhanced version presented in (Farnstrom et al., 2000). In the study (Bradley et al., 1998), individual data batch undergoes a series of steps including standard k-means (along with micro-clusters formed previously), primary and secondary data compression. There will be *k* clusters generated from the standard clustering step with each cluster represented by a CF tuple. Based on the *k*-clusters model, data in the current batch is grouped into micro-clusters representing discarded, compressed, and retained set, during the two-step data compression. Farnstrom et al. seek to reduce computational complexity and enhance

algorithm efficiency of scaleKM by removing the compressed and retained micro-clusters.

Scalable clustering algorithms which are not based on the CF and micro-cluster concepts of BIRCH are also found in the literature. For instance, (Hore et al., 2007) and (Wan et al., 2012) present fuzzy c clustering algorithms which are scalable to large data by processing them in batches, (Song & Wang, 2005) proposes a scalable GMM-based clustering approach in which GMM is trained on data batches. A main concern of these methods is the merging of two consecutive sets of clusters into single set, which can be complicated and erroneous if the involved data batches contain distinctive data distribution.

The scalable clustering algorithms discussed above are characterized by single or a few data scan, eliminating the need of transferring data from secondary storage device which is extremely expensive. However, the micro-clustering approach emerges as an exclusively flexible and effective means for data stream clustering. As many of the existing micro-cluster-based algorithms are designed by focusing only on the clustering accuracy, enhancement of computational efficiency is very much needed for continuous clustering applications.

2.6.2.3. **BIRCH**

In this section, we re-present the BIRCH algorithm which is originally proposed in (Zhang et al., 1996) by examining the CF vector, CF-tree structure, insertion of data into CF-tree, and maintenance of CF-tree. As has been mentioned above, CF vector is a tuple containing 3 statistics (*N*, *LS*, *SS*) extracted from the constituent data points of a micro cluster. The CF vector provides all necessary statistical information for clustering computations such as distance and cluster centers. In addition, the additive feature of

CF vector allows it to incrementally take in data points as well as to merge with other CF vector (i.e. micro cluster); merging of two micro clusters are sometimes needed for CF-tree maintenance, as will be discussed later.

CF-tree is made up of 3 types of micro clusters, namely cluster node, leaf node and interior node (known as nonleaf node in the original paper). These nodes are connected through parent-child relationship, as shown in Figure 2-2, interior nodes represent high level nodes containing leaf nodes as their children, and cluster nodes are the lowest level nodes which do not contain child node. Interior node located at the top of CF-tree is called the root node. To facilitate the increment of tree height, we define cluster nodes as level 0, leaf nodes as level 1, and interior nodes as level 2 and higher, which is the different from the original paper. Note that the CF vector of cluster node is computed directly from its constituent data points; whereas the CF vectors of leaf and interior node are obtained by adding the CFs of all children nodes. The maximum number of children node allowable in a leaf node (L) or an interior node (B) is determined by both the data dimensionality and page-size of computing system.

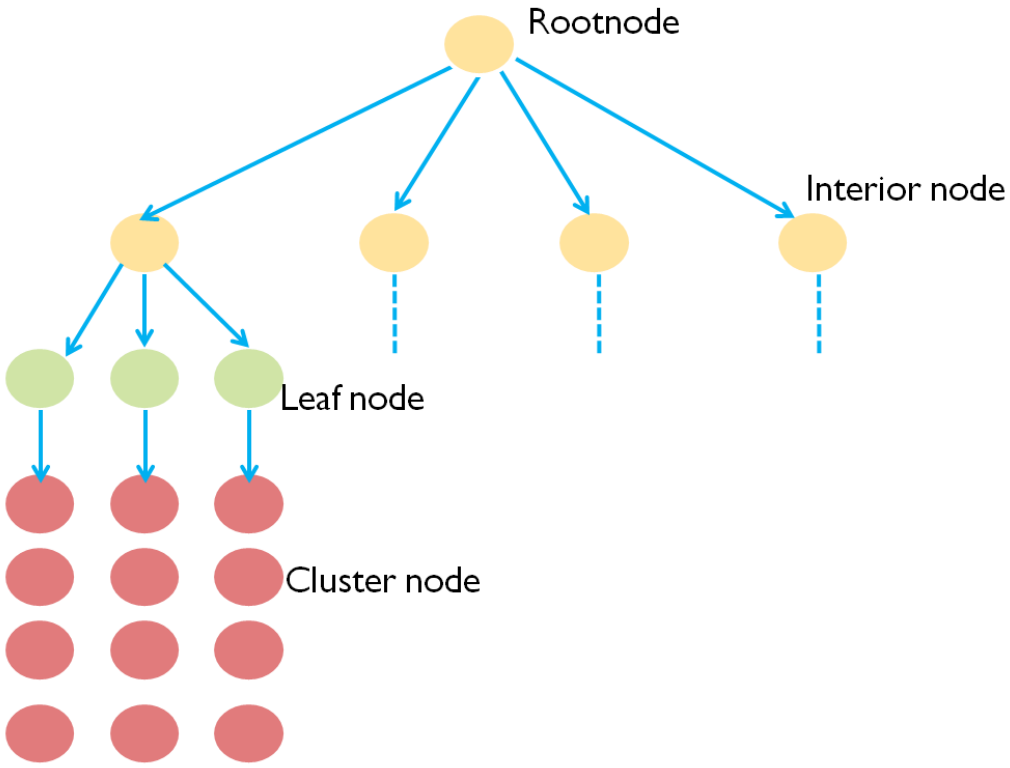

Figure 2-2: An example of CF-tree, where circles represent micro clusters and one-directional arrows link parent node to its children nodes.

Insertion of a data point *D*' into the CF-tree involves 2 key steps, which are identification of the nearest cluster node $C_{nearest}$ and updating of CF-tree with or without structure change. In the first step, the data point traverses through the CF-tree in a top-down manner, whereby the path taken is made up of the closest child nodes at every level, starting from the root node. Once it arrives at $C_{nearest}$, it needs to examine if the distance between *D*' and $C_{nearest}$ is smaller than the current threshold value. If this is the case, the CFs of $C_{nearest}$ as well as all its preceding parent nodes will be updated to reflect the adding of *D*'; note that there is no structure change involved. Otherwise, a new cluster containing single data point will be created as a new child sharing the same parent leaf node of $C_{nearest}$, provided there is empty child slot. It all child slots are occupied, splitting of the leaf node is required, whereby a new leaf node is created and the siblings of $C_{nearest}$ along with the newly created cluster node are re-distributed

between these two leaf nodes. Redistribution is achieved by locating two farthest children nodes which serve as the seeds in the two leaf nodes, and then the remaining children nodes will be assigned to the nearest seed. Splitting may propagate to higher levels following the same re-distribution procedure. If splitting reaches the root node, an additional level will be added.

A key characteristic of BIRCH algorithm is that it summarizes data points into the finest possible micro clusters, given a memory space *M*. As new data points are continuously inserted into the CF-tree, it may arrive at a state where *M* has been fully consumed. If this is the case, threshold value needs to be increased, and CF-tree will be rebuilt based on the modified threshold. It should be noticed that a larger threshold enables merging of nodes and hence resulting in a smaller size CF-tree. In this study, we apply the same heuristic approach given in (Zhang et al., 1996) for threshold increment, in which the new value is computed based on the distance of two nearest cluster nodes in the most crowded leaf node found in the CF-tree. CF-tree rebuilding involves treating individual cluster node as a new data point against the new CF-tree, where the new CF-tree initially consists of only the most left-hand branch of the original tree. During rebuilding, if a cluster node finds a closer path in the new tree before its original path, it will be either merged with another cluster node or included as a new child in the new path; otherwise it will stay in its original path.

Furthermore, BIRCH includes 2 additional features which are used for delaying node splitting and removing outliers during CF-tree rebuilding. The first property is achieved by keeping aside data points which demand for node splitting into a temporary variable *delay-split* and allowing insertion of subsequent data points; the second feature involves moving cluster nodes containing too few data points to a variable *outliers*. The

sizes of these two variables which are specified by the users affect both the quality and computational time of BIRCH, as they alter the sequence of data insertion and at the same time the variables need to be examined regularly for data re-insertion to the CF-tree.

Chapter 3. GPU-based Parallel Computing Technique

Historically, graphics processing unit (GPU) was specially designed for graphics rendering purpose. The major advantage of using GPU in mainstream computers is that it offloads massive amount of graphics-related operations from the central processing unit (CPU). Graphics rendering process involves a series of operations on every image pixel unit following graphic pipeline, which comprises of vertex generation and processing, primitive generation and processing, fragment generation and processing, and pixel operations. In order to display precise and real-time graphics on computer screen, the architecture of GPU is structured in a way that it is capable of processing enormous vector calculations efficiently, by incorporating massive amount of processing cores. In the context of computer science, vector calculation or vectorization refers to the use of a computer program that processes one operation on multiple pairs of operands concurrently. The simultaneously manipulated operands are commonly known as data parallelism in parallel computing study. It should be noted that conventional CPU consisting of one or a few processing cores fails to meet such graphical requirement, and therefore it requires GPU to work as a co-processor so as to relieve the computational burden.

The subsequent development of shader programming in GPU technology has unlocked the opportunities for non-graphical or general-purpose GPU applications. In traditional graphic pipeline, shader which calculates rendering effects on graphics hardware is hardcoded (i.e. fixed-function). To offer flexibility, programmable shader is introduced in modern GPUs, for instance, NVIDIA's GeForce 7800 GPU which comprises of 3 programmable shaders as depicted in Figure 3-1. Graphical effects such as position,

saturation, brightness, and contrast of all pixels or vertices can thus be constructed dynamically using variable shader algorithms. Taking advantages of the many processing cores and programmable shaders of GPUs, researchers have successfully extended the use of GPU to general-purpose computing. Some example applications are (Adams et al., 2007; Liu et al., 2009; Liu et al., 2006; Loop & Blinn, 2006; Shiue et al., 2005; Tarditi et al., 2006; Wong et al., 2007). However, these non-graphical applications of GPU are required to follow strictly the graphic pipeline, making it extremely user-unfriendly. Furthermore, managing and load balancing of multiple types of programmable shaders or cores is difficult and involving good knowledge in graphic programming.

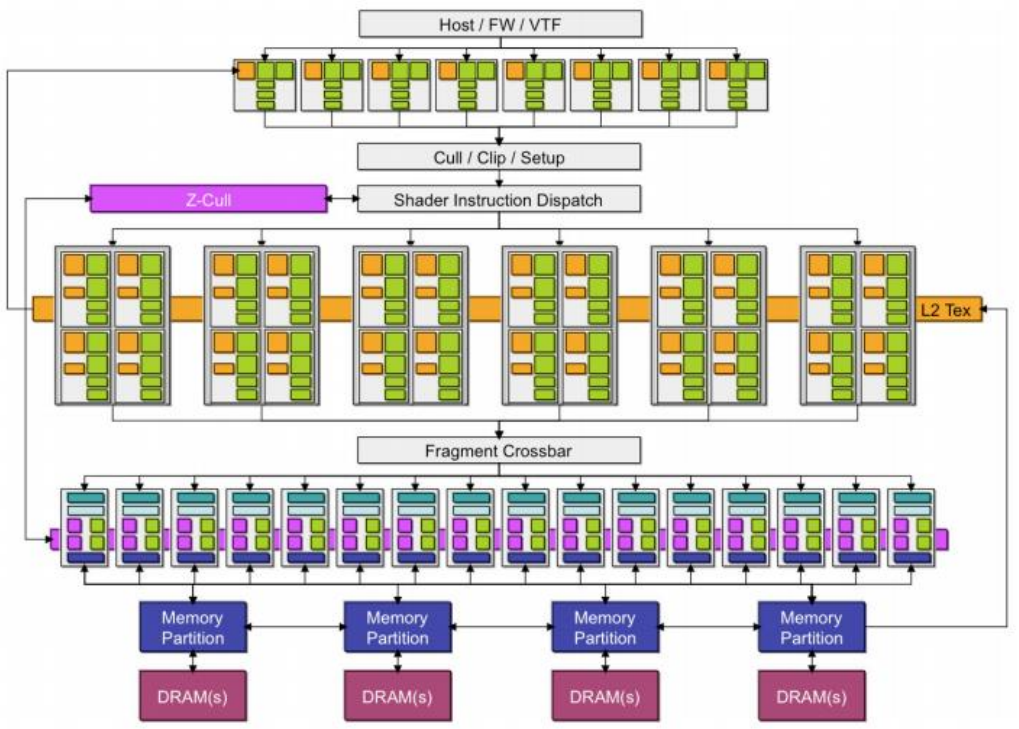

Figure 3-1: The 3 programmable engines of GeForce 7800 GPU developed by NVIDIA[1].

Graphics device developers such as NVIDIA and ATI introduce a new concept called 'unified shading architecture', where there is only single type of programmable

[1] Courtesy: (NVIDIA, NVIDIA's Fermi: The First Complete GPU Computing Architecture)

shader cores to accommodate all shader tasks. A typical example is the GeForce 8800 GPU developed by NVIDIA, as presented in Figure 3-2. Unifying of shader cores reduces programming complexity by eliminating the need for stage-by-stage balancing, and thus enables programmers to better focus on the algorithmic development. Nonetheless, programmers are still required to formulate the non-graphical GPU problem as a graphic rendering job by following the graphic pipeline and presenting data as vertex or texture information.

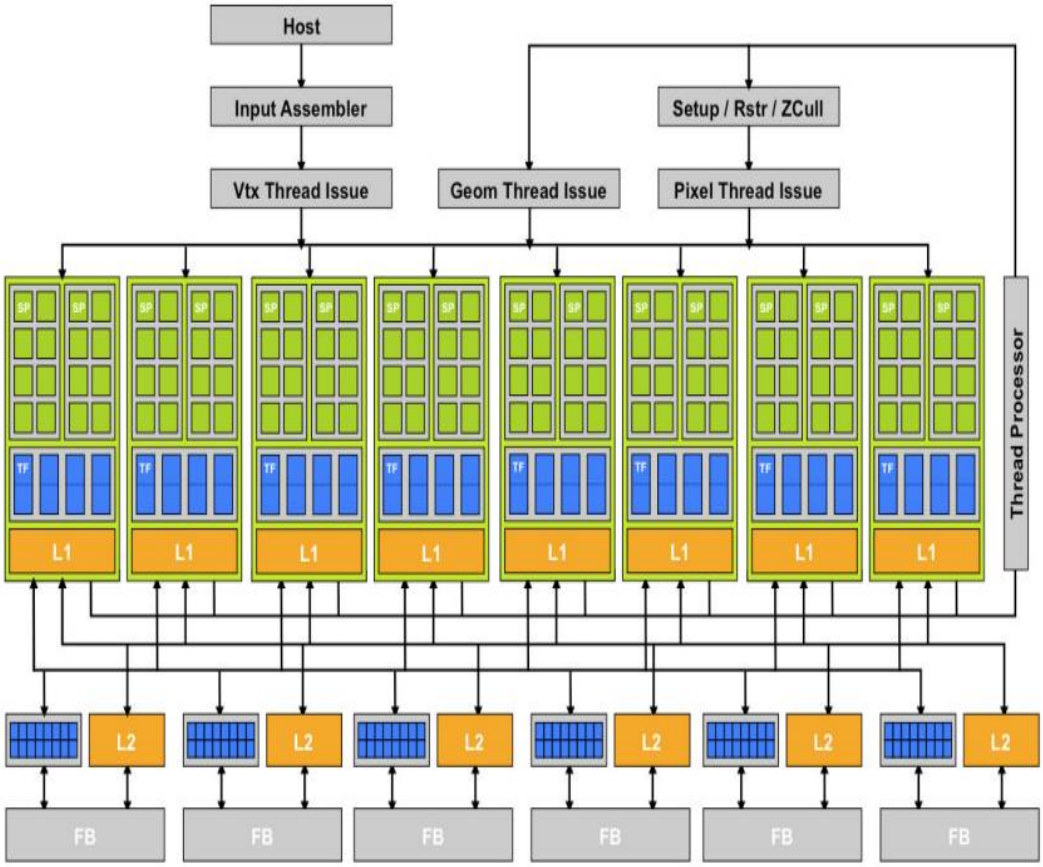

Figure 3-2: The unified shader architecture of GeForce 8800 GPU developed by NVIDIA[2].

A major breakthrough in the general-purpose GPU (GPGPU) computing is the introduction of a general-purpose programming model called Compute Unified Device

[2] Courtesy: (NVIDIA, NVIDIA's Fermi: The First Complete GPU Computing Architecture)

Architecture (CUDA) by NVIDIA in 2006. CUDA is the first C-based program development platform in GPGPU industry, which allows GPU to be treated as a many-core processor. Most importantly, CUDA provides a user-friendly development environment which requires zero knowledge of graphics operations, and thus decreases the learning curve and greatly improves computational efficiency. Within a few years after the introduction of CUDA, a wide spread of CUDA-based scientific computing applications have been reported; a number of classic examples can be found at (NVIDIA, GPU-ACCELERATED APPLICATIONS).

While CUDA programming is much easier and effective than shading programming, it requires adequate understanding on GPU architectural details and programming model characteristic for better utilization of device resources and development of highly efficient programs. In the following sections, we first investigate the hardware architecture of GPU in regards to key components such as processing cores, memory types, and its relationship with CPU. It is then followed by the discussion on important features of CUDA programming model, including the concept of threads, warp, thread block, and grid, efficient memory accessing patterns, and types of communication. Towards the end of this chapter, a number of CUDA applications in the area of optimization and pattern recognition will also be provided.

3.1. CUDA-enabled GPU Architecture - Hardware

A fundamental difference between GPU and CPU is that GPU is a computationally intensive device while CPU is a control-flow optimized device. As it can be seen from Figure 3-3, GPU devotes many of its transistors to Arithmetic Logic Units (ALUs), while CPU allocates a significant amount of transistors to flow control and data caching. Given a massive amount of ALUs, GPU possesses a much higher arithmetic

computing power than its CPU counterpart. However, the deficient control-flow capability of GPU has limited it to be a co-processor instead of the host processor of an application. In other words, only certain parts of an application are worthwhile to be ported to GPU, while CPU is still needed for controlling the overall process flow and also dealing with strictly sequential computations. The resulting program which is managed jointly by both CPU and GPU is called a hybrid CPU-GPU algorithm, or sometimes termed as GPU-based algorithm for brevity. On the contrary, a program that is executed exclusively on CPUs is called a sequential or serial program.

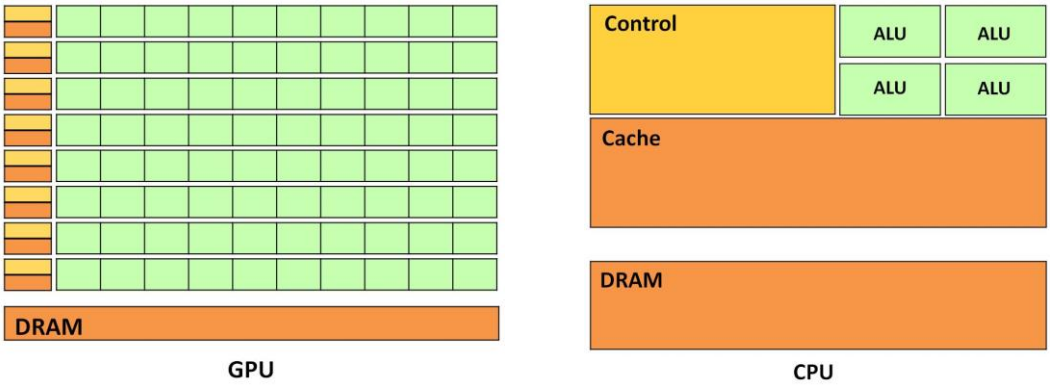

Figure 3-3: Comparison of GPU and CPU architecture[3].

The massive amount of processing units and low level control-flow units of GPU makes it especially suitable for realizing data parallelization. In the context of parallel computing, data parallelism refers to the concurrent computations on many different data elements. Particularly, GPU execution fits into the Single Program Multiple Data (SPMD) model whereby the same program (or called data parallel algorithm) is executed on many data simultaneously. A trivial example is the summation of 2 matrices, where the same adding operation (i.e. program) is executed on all matrix elements. By means of data parallelization, GPU is capable of achieving an enhanced

[3] Courtesy: (NVIDIA, NVIDIA CUDA C Programming Guide Version 4.2, 2012)

overall efficiency, despite of the suboptimal performance of individual operations due to deficiency in control-flow operation, as illustrated in Figure 3-4. Amdahl's law defines the theoretical amount of efficiency improvement achievable by a parallel computing system, as given below,

$$Theoretical\ Speedup = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1 - B)\right)} \quad (3\text{-}1)$$

where, $T(1)$ and $T(n)$ denote the time taken by the sequential and hybrid program, respectively, and the 1 and n inside the parenthesis indicate the number of data element being processed concurrently at the same time; B is the fraction of algorithm that is strictly sequential.

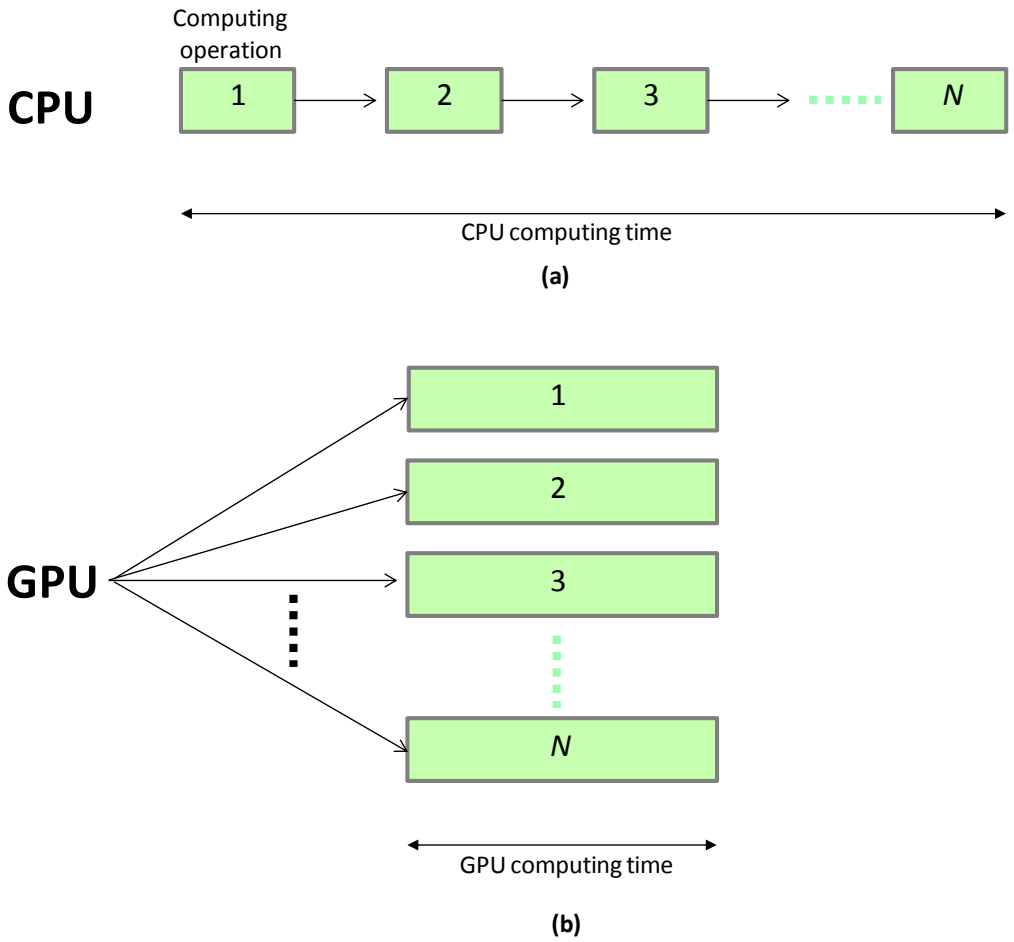

Figure 3-4: Data processing model of (a) CPU, and (b) GPU computing system.

The development of an efficient data parallel algorithm is not only about identifying data parallelism, it is also required to ensure GPU resources is fully utilized to achieve high arithmetic computing power. It is therefore important for GPU programmers to fully understand the detailed organization and limitations of various GPU components. In a CUDA-enabled GPU device, the processing cores or streaming processors (SPs) are organized into multiple streaming multiprocessor (SM) units, as shown in Figure 3-5. It should also be noted from Figure 3-6 that SPs within the same SM unit share several computational resources, e.g. the instruction dispatch units, warp schedulers, shared memory, cache, and register file etc. This sharing of instruction dispatch units and warp schedulers implies that the computations on these SPs are

executed in parallel, whereby the same instructions are broadcasted to multiple SPs. Also, communication among SPs can be done through the shared memory. Hence, SMs and SPs together represent two-level of data parallelism in CUDA-enabled GPU. In developing a CUDA-based data parallel algorithm, it is crucial to adequately map data operations to these two-level processing units in such a way that both SPs and SMs are optimally utilized during the entire course of GPU computing. For instance, a data parallel program which assigns a massive amount of data operations to single SM will perform poorly, as SPs in all other SMs are not used. A further point is that the amount of device resources e.g. the number of SPs in a SM, the size of shared memory and register file in each SM, varies with the GPU architectural models which are specified as compute capabilities such as 1.0, 1.1, 2.1 by NVIDIA.

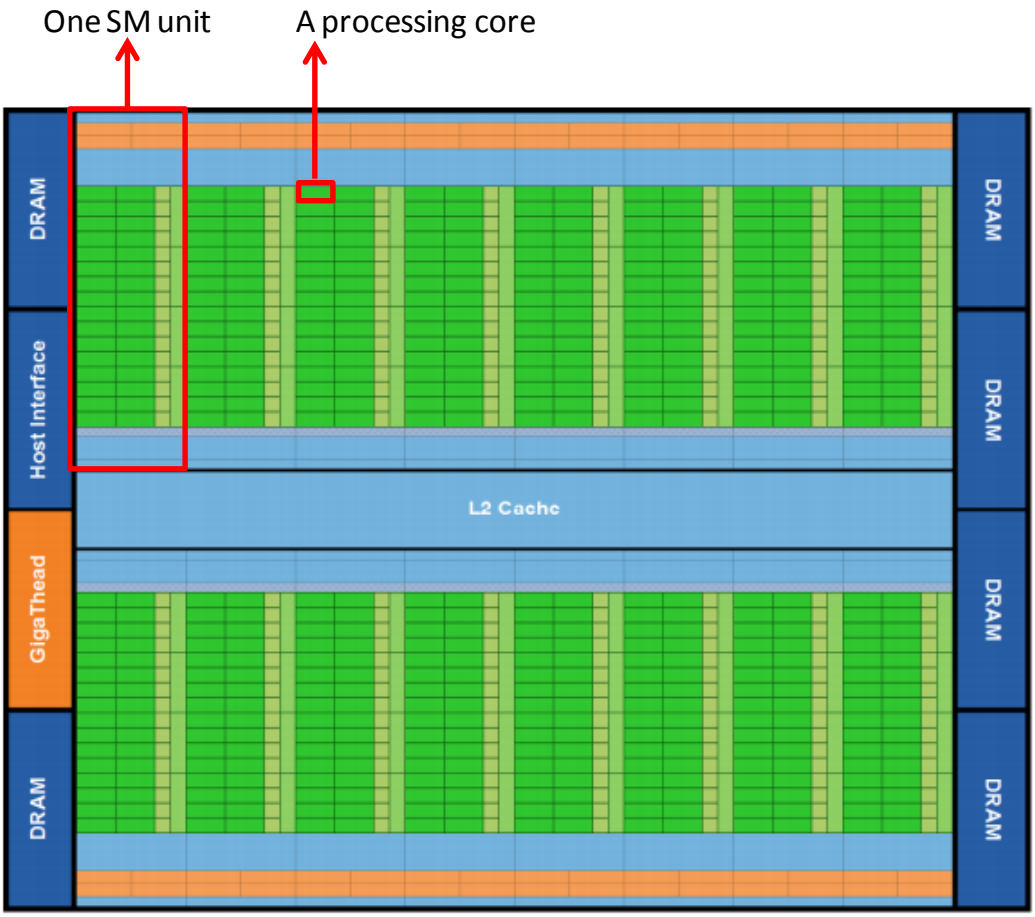

Figure 3-5: An architectural overview of a NVIDIA's GPU device in the Fermi series; a closer view is presented in Figure 3-6[4].

[4] Courtesy: (NVIDIA, NVIDIA's Fermi: The First Complete GPU Computing Architecture)

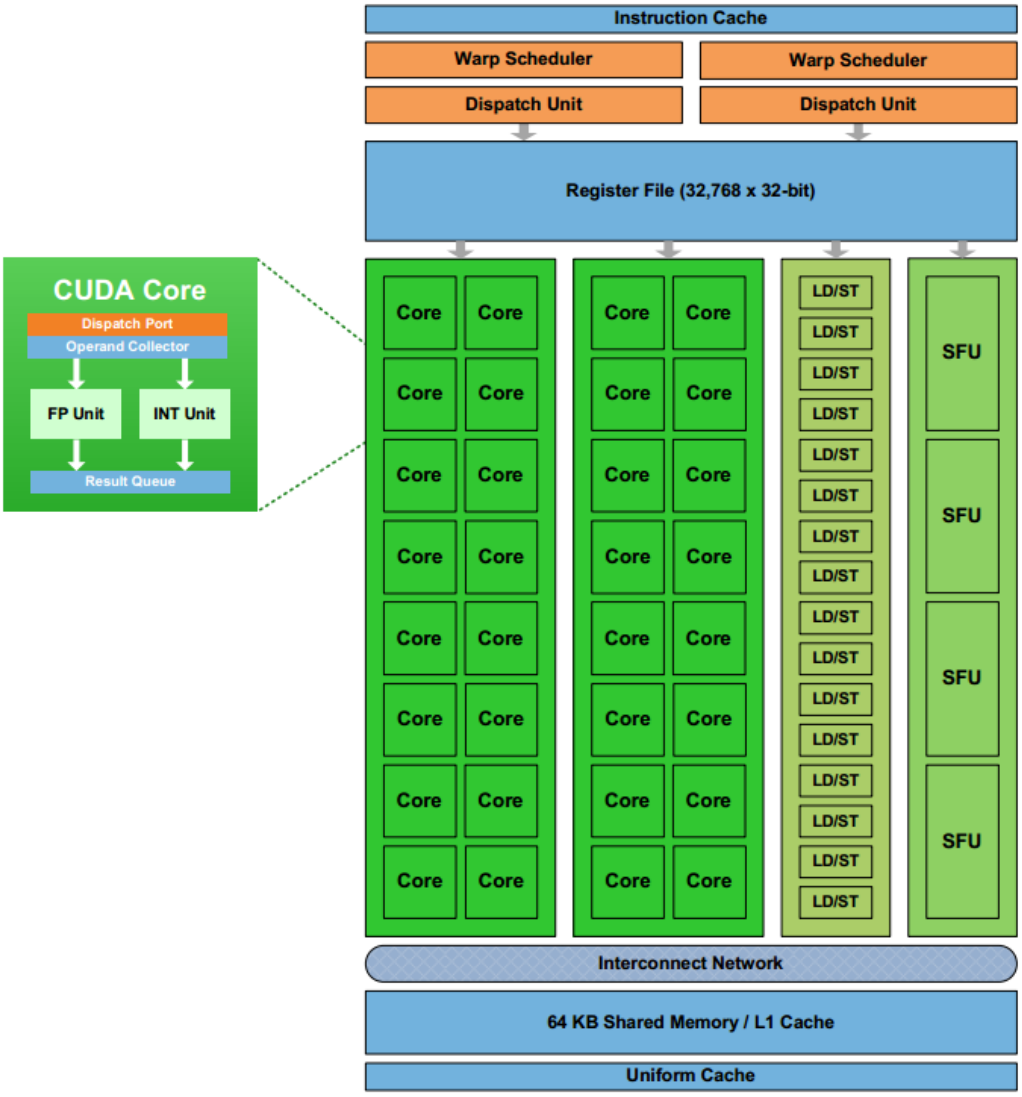

Figure 3-6: The key components contained in a SM unit of Fermi series GPU[5].

3.1.1. **Memories**

Besides the shared memory mentioned early, there are several more memory types provided in CUDA-enabled GPUs, namely global memory, constant memory, texture memory, local memory, and register. Essentially, these memories are different in terms of memory space, latency, accessibility, read-write-ability, and location, as summarized in Table 3-1, thus they are suited for different data storage purposes.

[5] Courtesy: (NVIDIA, NVIDIA's Fermi: The First Complete GPU Computing Architecture)

Global memory which is also known as DRAM, as presented in Figure 3-5, takes up the largest memory space among all. It can be used for storing both input (read) and output (write) data, which are usable and writable by all data operations in a data parallel program grid as well as the host CPU; the concept of grid, block, thread, and application will be introduced in Section 3.2. Due to its off-chip position, it involves high memory latency especially during data reading. In writing data to global memory, it proceeds to the next instruction immediately after calling for the store operation without waiting. Memory latency ranges from 400 to 700 clock cycles depending on the memory accessing pattern. The lower end can be achieved by the desirable coalesced memory access, which will be described in details in Section 3.2. Constant and texture memory are read-only memories which have limited memory space. Despite of their off-chip position, relatively low memory latency is achievable if specific memory access patterns are satisfied, which are data broadcasting and spatial locality, respectively. Such memory accessing patterns enables efficient use of constant cache (i.e. the uniform cache shown in Figure 3-6) and texture unit cache, which will be discussed more in the subsequent section.

Similar to CPU-based HPC computing system, there is memory copying overhead costs incurred during GPU computing. As GPU is prohibited to access host memory directly, input data needs to be first copied from host to GPU global or constant or texture memory before any GPU computing can be performed. Besides, upon completion of GPU computation, the host needs to copy the results from GPU global memory back to host memory for further computations. The GPU and host memories are connected through a Peripheral Component Interconnect (PCI) bus, thus the data transfer rate is limited by hardware specifications. An illustrative example is given in Figure 3-7 showing different directions of memory copy and the associated

memory bandwidths. In general, GPU memory has a higher bandwidth than CPU memory, while the memory transfer through PCI bus is the slowest. Hence, data transfer between host and GPU should be minimized at all time, otherwise it may offset the performance benefits of GPU computing.

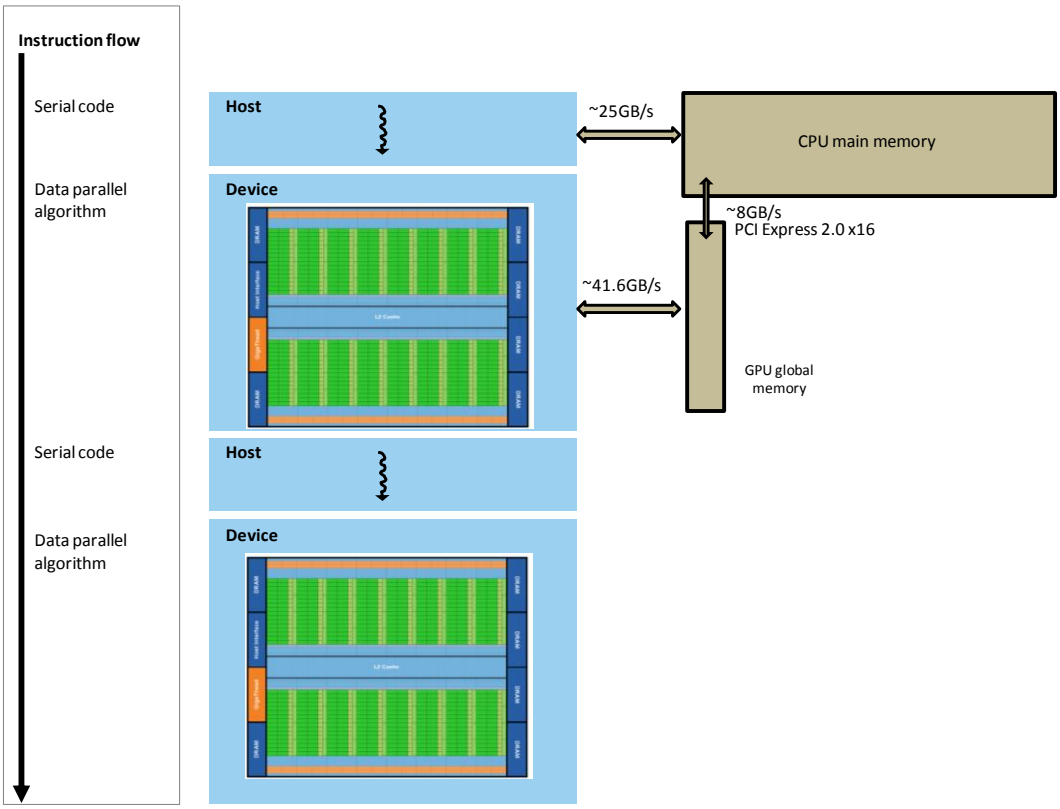

Figure 3-7: Instruction flow and memory copies involved in a hybrid CPU-GPU application; memory bandwidth values are based on a PCI express bus 2.0x16, Intel Xeon W3670 workstation, and a Quadro 2000 GPU.

Next, we discuss the two on-chip memories, shared memory and register. Shared memory serves as a communication platform within a SM unit, in which data computations executed on the same SM can exchange information e.g. intermediate results with each other. To fulfill the requirement of information sharing, it must be read-write-able. It is a very fast memory which takes about 4 or more clock cycles for memory accessing, and this is 2 orders of magnitude faster than the off-chip memories. Similar to the abovementioned memories, the lower end of latency is achievable by

satisfying a desirable memory access pattern, which is non-bank-conflicting access for shared memory as it will be analyzed in subsequent section. As the size of shared memory is rather small, which is at most 48 KB, its usage should be limited for data which is accessed or modified frequently during the operations of a data parallel algorithm. Note that the size of shared memory in GPUs with compute compatibility of 2.X or above is configurable to be 48 or 16 KB. On the other hand, register is the fastest CUDA memory which allows memory accessing to occur at almost zero cost. However, it is limited to extremely small memory space, and it is the smallest among all memory types. Besides memory space limitation, register is restricted for storing data which is relevant to single data operation or a thread as it will be discussed later. It is essential to ensure that the memory space allocated for register not exceeded, otherwise it will cause spillage to expensive local memory. Memory latency involved in accessing local memory is as high as global memory.

Table 3-1: Characteristics of the 6 CUDA memories; the values shown are relevant to GPU with compute compatibility of 2.X.

Memory type	Features					
	Memory size	Latency	Accessibility	Read-write ability	Location	Lifetime
Global	~1GB or more	400 - 700 clocks	Grid & host	Read-write	Off-chip	Application
Constant	64 KB	50 - 500 clocks	Grid	Read-only	Off-chip	Application
Texture	Max width for 1D array: 32768 or Max width for 2D array: 65536x65535 or Max width for 3D array: 2048x2048x2048	250 – 500 clocks	Grid	Read-only	Off-chip	Application
Local	512 KB per thread	400 - 700 clocks	Thread	Read-write	Off-chip	Thread
Shared	48/ 16 KB	>= 4 clocks	Block	Read-write	On-chip	Block
Register	32K 32-bit per SM	0 clock	Thread	Read-write	On-chip	Thread

In this work, all GPU computing is done using a NVIDIA's Fermi Quadro 2000 GPU device with compute capability of 2.1. Computing resource characteristics of the device are presented in Table 3-2. Note that computational power is quantified by number of floating-point operations per second (FLOPS) or giga FLOPS. The GPU contains full-featured processing units in term of full IEEE 754-2008 support for both

single and double precision floating-point operations. The device also includes special function units (SFUs) for transcendental, reciprocal, and square root operations, as its CPU counterpart. On top of the procedural C code, it offers object-oriented C++ programmability. Such GPU is suited for accurate scientific computation.

Table 3-2: Specifications of NVIDIA's Quadro 2000 GPU device.

Device Resources	
SMs	4 per device
SPs	48 per SM
SFUs (Special Function Unit)	8 per SM
Registers (32-bit)	32,768 per SM
Memory size	1024 MB (GDDR5, 128-bit width)
Memory Latency	400-600 clocks
Memory Bandwidth	41.6 GB/s
Constant Memory	64KB per device
Shared Memory	48 KB per SM
Warp Scheduler	2 per SM
Peak computational power	Single Precision: 480 GFLOPS Double Precision: 240 GFLOPS
Caches	
Constant Cache	8 KB per SM
Texture Cache	6-8 KB per SM

3.2. CUDA Programming Model - Software

In CUDA programming, threads are the abstract processing elements which map the massive amount of data parallelism to processing units. During the execution of a data parallel algorithm, a CUDA thread will be created and assigned to an individual data operation, thus *N* threads will be needed if the algorithm is operated on *N* different data elements. Depending on the complexity of the computational task, cooperation among different data operations is sometimes required. Considering the previously discussed matrix summation example, if the resultant matrix is further reduced to a column vector through row-wise summation, data operations along the same row are

required working together to determine the summed value. To allow for cooperative operation while reserving massive parallelism, CUDA threads are organized into blocks and grid, as illustrated in Figure 3-8; where threads and blocks represent the two-level data parallelism. Thread configuration is the term used for describing thread organization, and it consists of two components which are the grid size (i.e. the number of blocks in a grid) and block size (i.e. the number of threads in each block). Threads within the same block are granted certain degrees of dependency, while threads at different blocks are strictly independent; more detailed discussion will be given in later paragraphs. Note that grid defines the scope of a data parallel program or called CUDA kernel or function, thus all threads inside a grid execute the same program. As every thread in a block and every block in a grid is assigned with a unique identifier, it is straightforward to compute a globally (i.e. grid-level) unique thread identifier for associating with relevant data elements (i.e. its share of work) or specific operational path.

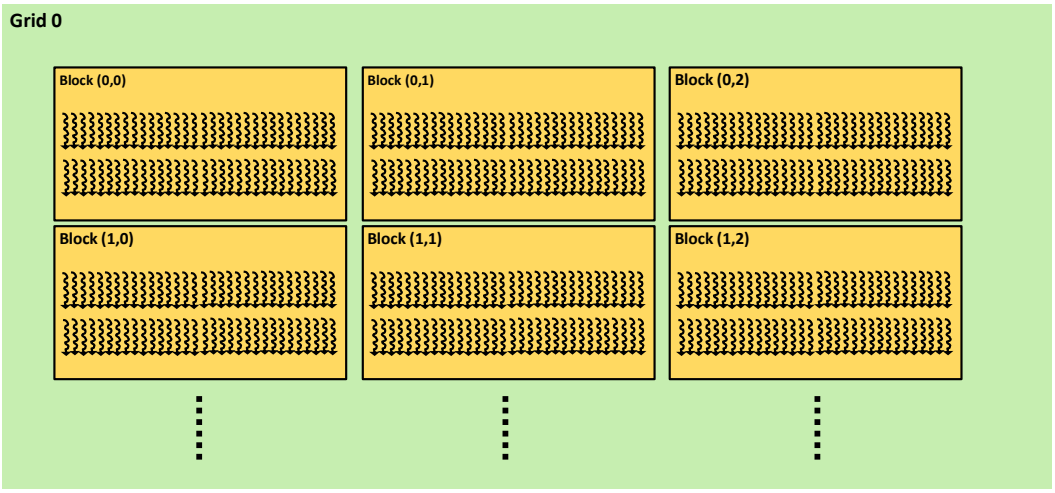

Figure 3-8: An example of CUDA threads configuration, demonstrating the relationship among threads, blocks and grid.

3.2.1. CUDA blocks execution

The mapping of abstract processing threads to actual processing units in GPU device is presented in Figure 3-9 and Figure 3-10. As shown in Figure 3-9, the assignment of CUDA blocks to SMs is done randomly. Depending on the computing resource requirement for each block as well as the resource limitation of SM, there could be one or more blocks managed by a particular SM at one time. However, it is important to ensure that computing resource required by a CUDA block must be bounded such that at least one block can be fitted into a SM. Once the operations of a CUDA block are completed, a new block waiting in the queue will be assigned to the SM. All SMs of a GPU device are always occupied unless there is no more unprocessed block in the grid.

3.2.2. CUDA threads execution

Now, we consider the execution of a particular CUDA block that is assigned to the first SM, as given in Figure 3-10. A warp is defined as 32 (or 16) consecutive threads in a block, when a GPU device with compute capability of 2.X (or 1.X) is used. For instance, a block consisting of 128 threads can be viewed as 4 warps, which correspond to threads with identifier in the range of [0, 31], [32, 63], [64, 95], and [96, 127], respectively. Since warp is the smallest execution unit in CUDA programming model, it is advisable to have CUDA blocks containing number of threads that is multiples of warp size, so as to avoid resource wastage. As it can be seen from Figure 3-10, warp schedulers look for available warp randomly from the assigned CUDA block, and then allocate with suitable instruction. During this process, there are two issues need to be addressed. First, warps that are available for execution must have their previous operation completed; note that available warps are different from resident warps which

are simply every single warp in the assigned CUDA block. In general, the number of available warps tends to increase if there are more resident warps. Hence, thread configuration which exhibits large number of resident warps is usually more favorable. Secondly, the required instruction units which can be SPs, or SFUs, or memory load or store units, must be unoccupied and free for use.

An important characteristic of warp execution is that threads within a warp are self-synchronized and thus required to follow strictly the exact operational path. Such execution paradigm fits to the Single Instruction Multiple Thread (SIMT) model as defined by NVIDIA. Supposing there are *P* conditional paths given in the CUDA program and each thread within the warp takes a different path, the entire warp will need to walk through the 32 operations sequentially. Hence, it is crucial to ensure that all threads in a warp taking the same or a limited number of paths. On the other hand, threads in different warps, regardless if they belong to the same CUDA block, can take different paths at no additional cost.

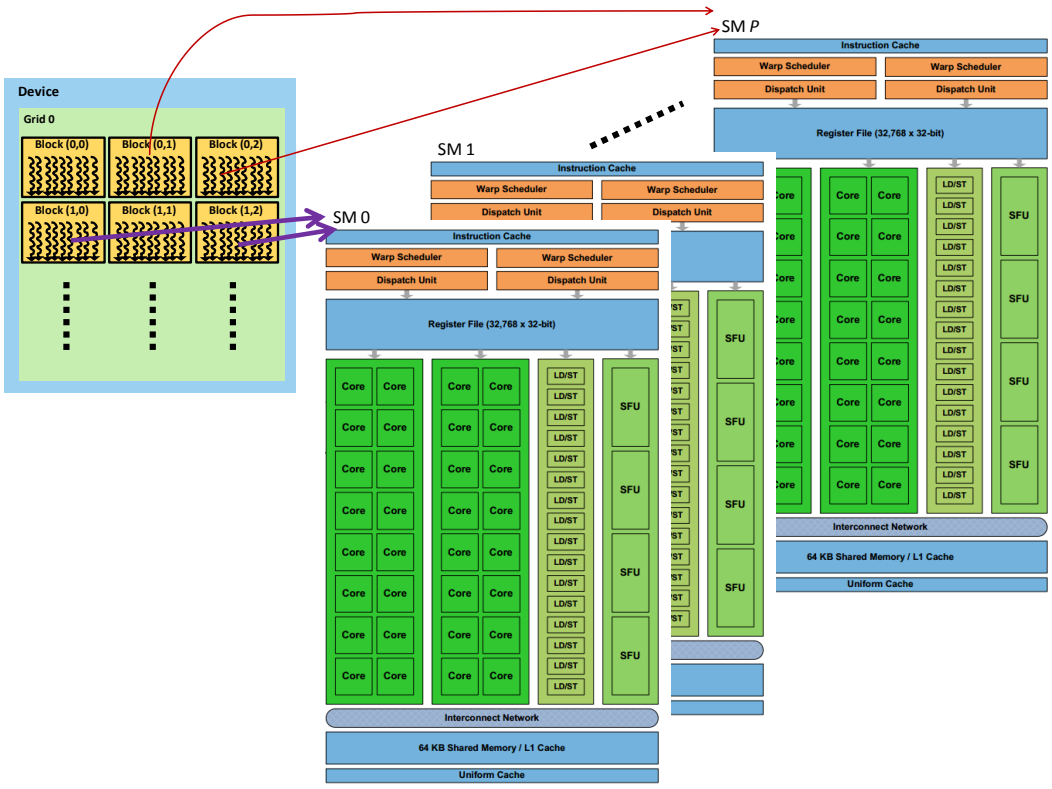

Figure 3-9: Mapping of CUDA blocks to SM units in a GPU device.

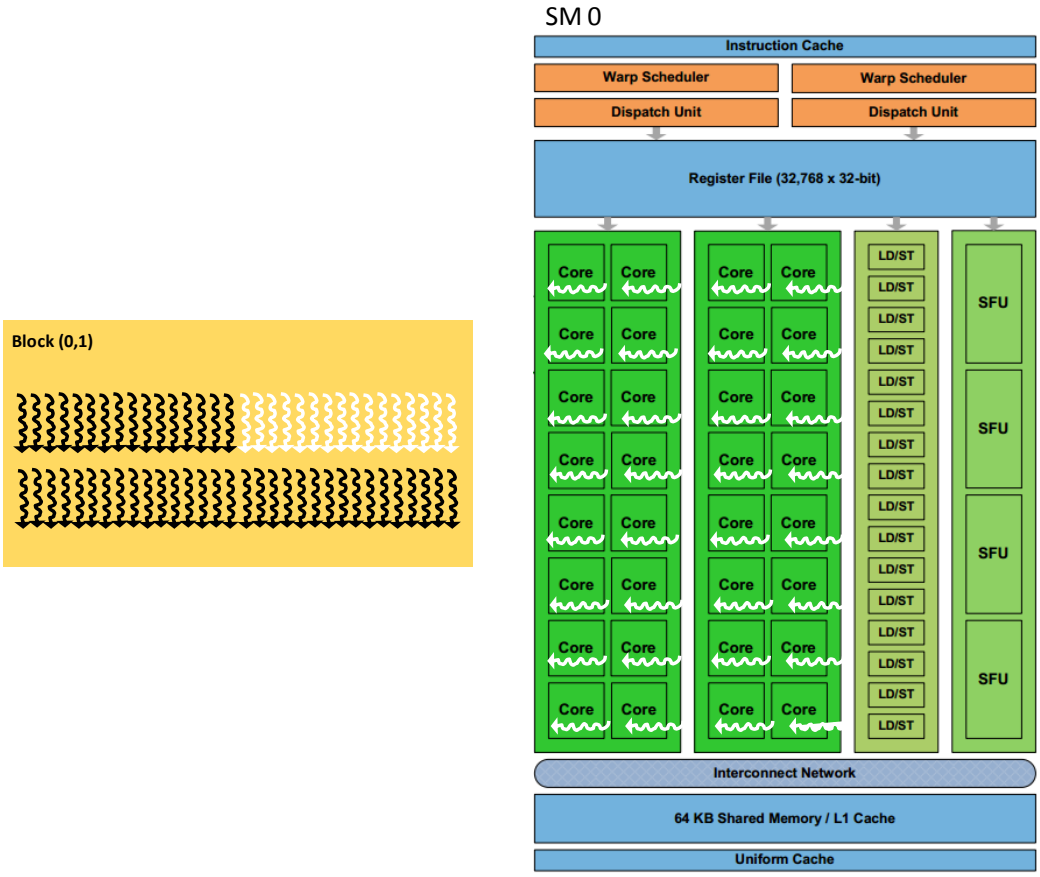

Figure 3-10: Mapping of CUDA threads to SPs in a SM.

As it has been pointed out early, the allocation of threads inside the same block allows for more useful work to be done. This can be attributed to the availability of 2 valuable features, which are shared memory and synchronization barrier. As discussed in Section 3.1, shared memory serves as a communication platform for threads within the same block to exchange information in the middle of CUDA kernel execution. On the other hand, synchronization barrier provides a stop point to align all threads in a block. This synchronizing feature is particularly important when the correctness of subsequent computation is affected by the current operation of other threads in the block. Since warp execution is self-synchronized, CUDA block containing only a warp will behave in the same way. However, the synchronization of threads within a block incurs 4 or more clock cycles, and it is usually much longer especially for

large CUDA block, due to the need of waiting. Hence, cautions must be taken while applying this feature. In contrast, thread operations at different blocks cannot be synchronized, and the only way to do that is to split the computing application into several CUDA kernels or programs. Table 3-3 presents the software resource limitations relevant to a NVIDIA's Quadro 2000 GPU device.

Table 3-3: Programming model of NVIDIA's Quadro 2000 GPU device.

Programming Model	
Warp size	32 threads
Max block size	1024x1024x64 threads
Max grid size	65535x65535x1 blocks
Max resident block per SM	8
Max resident warps per SM	48
Max resident thread per block	1536
Maximum 32-bit registers per thread	63 bytes
Kernel Size	2 million instructions max

3.2.3. **Memory access**

In regards to memory accessing, warp is the most relevant execution unit in determining memory latency for various types of memory; more details can be found at NVIDIA's website (NVIDIA, CUDA TOOLKIT DOCUMENTATION). Global memory can be viewed as memory chunks, where each chunk consists of 32 contiguous 4-bytes words. If a data request operation associated to a warp involves one such memory chunk, single memory transaction is needed, as demonstrated at the top example of Figure 3-11. This memory access pattern which is called coalesced access is the most favorable for global memory, as it requires only one memory transaction and thus reduces memory latency. There are cases where the data requested by a warp fall into different memory chunks, thus requiring multiple memory transactions, like the center and bottom examples in Figure 3-11. It should be noticed that some parts of the data obtained from each memory transaction are not used, and thus resources are wasted.

Constant and texture memories are cached memories which require special accessing patterns to achieve low memory latency. Constant cache is designed for data broadcasting, thus it is exclusively used for situation where the threads in a warp ask for the same data. Texture cache is optimized for accessing data exhibiting spatial locality, as shown in Figure 3-12.

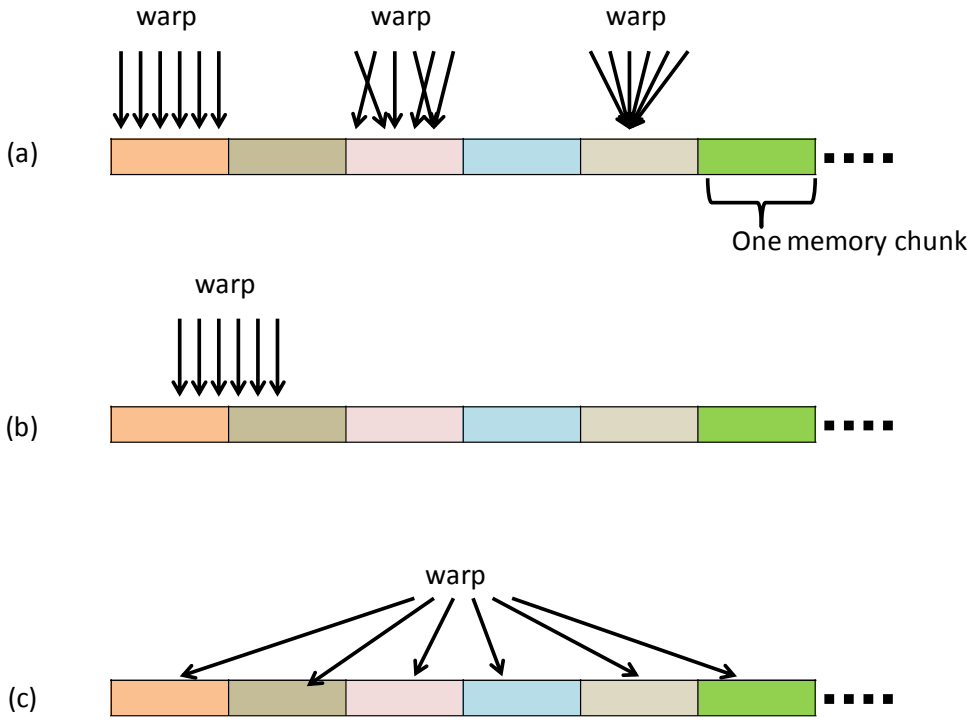

Figure 3-11: A small warp containing only 6 threads is used to illustrate the number of memory transactions under 3 different situations, where (a) one, (b) two, (c) six, transactions are required, respectively.

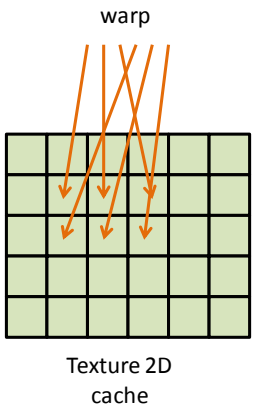

Figure 3-12: An example of texture memory access showing spatial locality characteristic.

On the other hand, memory accessing to shared memory is more complicated and compute capability dependent. In this work, all GPU computing is done on a Fermi GPU device with compute capability 2.1, so the following discussion on shared memory is focused on GPU of compute capability 2.X. Shared memory is made up of an array of 32 memory banks, in which successive 32-bit words are stored at successive banks. An important feature is that data in the 32 banks can be accessed simultaneously, within a warp. In general, we want to avoid bank conflict which serializes memory access. Bank conflict occurs when there is more than one thread in a warp attempt to access the same bank, for example if there are two threads accessing the same bank, it is called 2-way bank conflict like example (e) in Figure 3-13. Supposing the 32 threads in a warp access to different banks during a data request, there will be no bank conflict and minimal memory latency is achievable. As shown in Figure 3-13, examples (a), (d), and (f) satisfy this criteria. However, in a special case where multiple threads access to the same 32-bit word in a memory bank, there will also be no bank conflict; such as the examples (b) and (c).

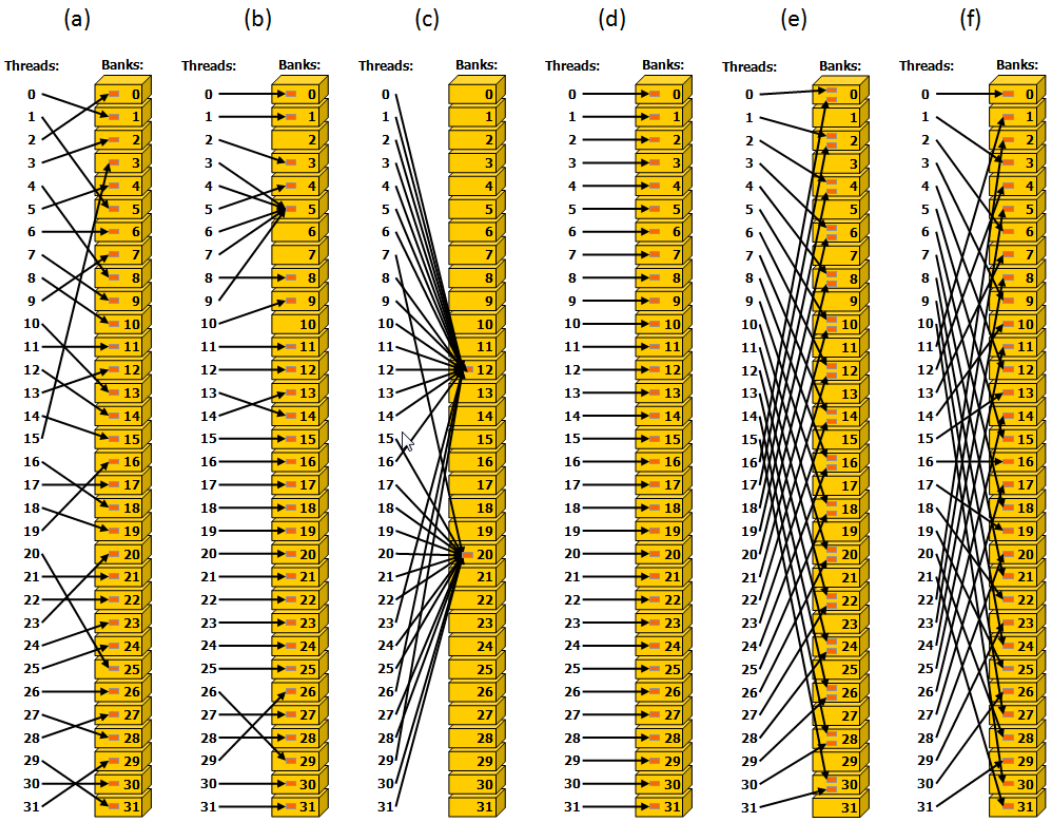

Figure 3-13: Examples of shared memory access; (a) random access; (b) random access with broadcasting; (c) broadcasting; (d) orderly access; (e)&(f) strided access; where (a), (b), (c), (d), (f) have no bank conflict, (e) generates 2-way bank conflict[6].

3.3. CUDA Programming Issues

CUDA software package which is freely downloadable from NVIDIA's website (NVIDIA, NVIDIA CUDA ZONE) mainly consists of programming toolkits (e.g. user manual and profiler), device driver, compiler and Application Program Interface (API) functions. In particular, CUDA software package of version 4.2 64-bit is employed in this work (NVIDIA, CUDA Toolkit 4.2 - archive). At CUDA platform, software program development is based on standard programming languages, including C/C++ and Fortran. Here, we focus on the use of CUDA C/C++. A CUDA program developed based on CUDA C/C++ is compiled with NVIDIA's LLVM-based C/C++ compiler (NVCC) which comes with the

[6] Courtesy: (NVIDIA, CUDA TOOLKIT DOCUMENTATION)

software package. As a hybrid application consists of both serial and parallel codes, a key role played by the NVCC is to identify, separate and send the two parts of codes to CPU host and GPU, respectively.

Due to the fundamental difference between CPU and GPU in terms of hardware features and programming model, extensions to standard C/C++ are needed. These include function type qualifier, variable type qualifiers and built-in variables; detailed discussion is provided in (NVIDIA, NVIDIA CUDA C Programming Guide Version 4.2, 2012). Function (or variable) type qualifiers specify the execution (or storage) location of a particular function (or variable), either on the host or GPU device; while built-in variables are used to indicate thread and block identifiers as well as the size of block and grid. For instance, a CUDA kernel has to be declared with __*global*__ function qualifier along with <<<…>>>; the former notifies NVCC that this is a GPU function, and the latter contains the execution configuration. The execution configuration essentially refers to the thread configuration employed during a particular run of the CUDA kernel. Moreover, API functions included in the CUDA package provide a simple means for data transferring, and thus simplifying the process of porting CPU sequential code to GPU.

3.3.1. **Guidelines for developing CUDA-based GPU program**

A standard set of procedures which serve as broad guidelines for developing CUDA-based GPU application is summarized in Figure 3-14. First of all, potential GPU tasks are identified based on two main criteria, which are high computational cost, and reasonably rich amount of data parallelism relative to the amount of data transfer required between host and GPU. The next step is to design and build CUDA functions for the selected GPU tasks, whereby good knowledge of GPU architecture and parallel programming is required. In general, there are a number of design factors need to be

considered, and these include the availability and suitability of existing GPU library functions, the mapping relationship of tasks and CUDA kernels, choice of memory types, optimum thread configuration, and the applicability of various optimization techniques. In the following subsections 3.3.2 and 3.3.3, some of the primary GPGPU operations and widely used optimization techniques are elaborated. The developed GPU program and the entire CPU-GPU application is firstly tested for computational accuracy and result correctness, after which computational performance is measured using some kind of profilers such as NVIDIA Nsight Visual Profiler. If the resulting computational speedup is not satisfactory, further improvement could be achieved by searching for additional tasks for GPU implementation or refining the design of CUDA kernels. Application of the standard GPGPU programing procedures shown in Figure 3-14 is demonstrated in 3 case studies throughout Chapter 3 to Chapter 5.

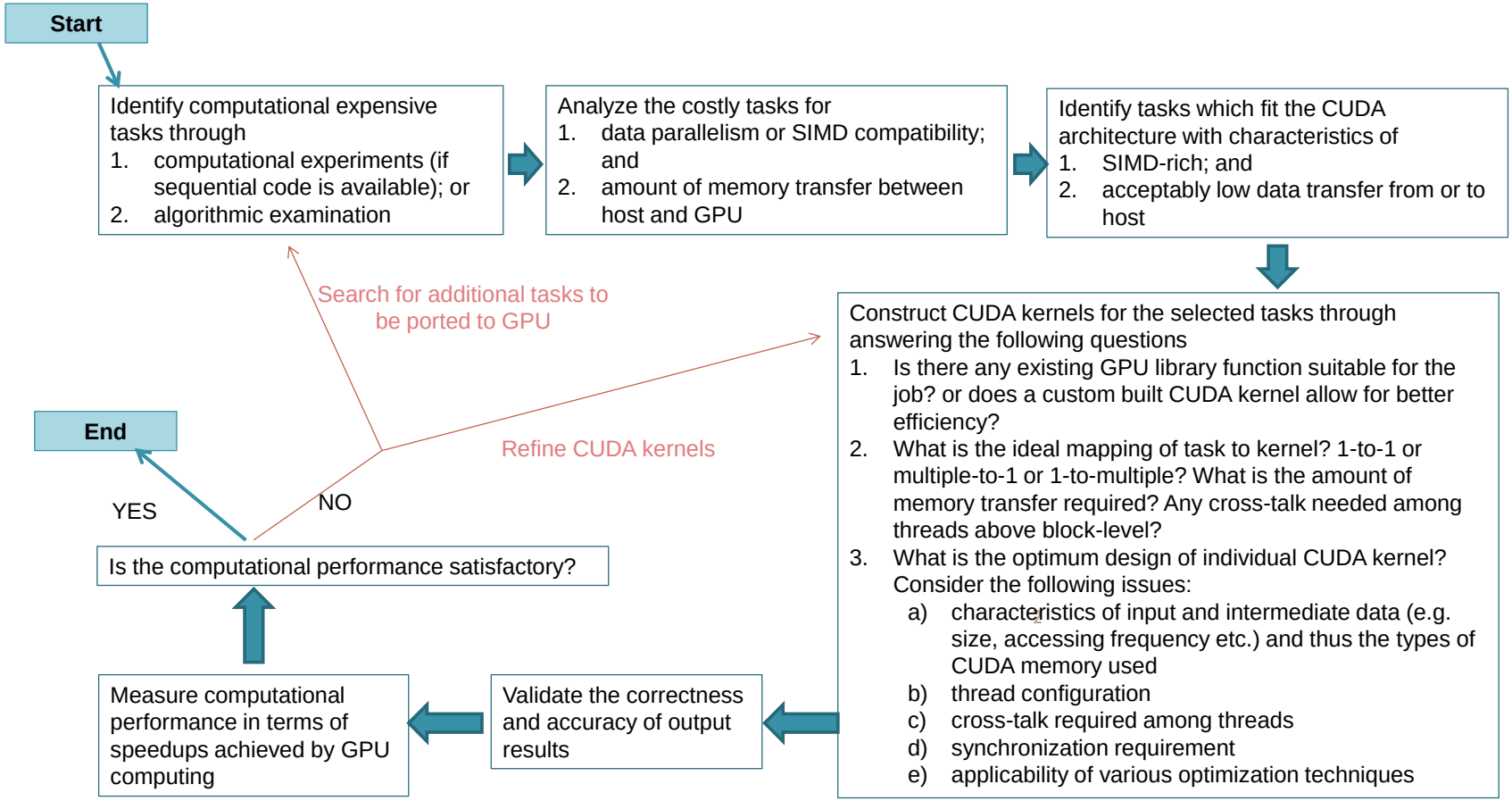

Figure 3-14: Proposed procedures of developing a CUDA-based GPU application.

3.3.2. **CUDA kernel design**

Programming knowledge and experience on one platform may not be directly useful to other platforms, particularly when the underlying programming models are disparate as in the case of transforming sequential code to parallel code. In view of the intensive programming effort of developing CUDA application, we recommend considering only computational expensive tasks for GPU computing. The simplest implementation of GPU computing is to directly apply functions from GPU-based libraries, such as CUBLAS, CULA, MAGMA, and Thrust libraries. Both CUBLAS and CULA are GPU-accelerated linear algebra libraries, where the former is the GPU-version of BLAS library; MAGMA library provides GPU-based dense linear algebra functions, similar to the sequential LAPACK library; and Thrust resembles the C++ Standard Template Library (STL). However, these standard library functions are insufficient for addressing computing problems that are complex. Real-world computing applications are always more than linear algebras, array sorting or searching. Hence, application-specific GPU programs need to be developed for optimal computational performance. A CUDA programming book written by Jason et al. (2010) provides simple CUDA kernel examples which are useful for beginners.

There are a number of primary GPGPU operations which serve as the building blocks of a GPU program; most relevant ones are map, reduce, scatter and gather, scan, stream filtering, and sort (Owens et al., 2007). These are data-dependent operations offering data parallelism for GPU computing.

Map: involves application of the same function to different data elements, e.g. given an array of input data [1, 2, 3, 4, …, 10] and a function $f(x + 2)$, the mapping operation results in output data of [3, 4, 5, 6, …, 12].

Reduce: involves computations which generate single element or a smaller array from a larger size input array, e.g. finding summed, maximum, or minimum values. On the CUDA platform, shared memory can be used to partially reduce the input array to intermediate summed array which has the same size as the block. An example is illustrated in Figure 3-15, where the input array of 20 elements is partially reduced to an intermediate array of 6 elements. The partially summed array can then be reduced further to single value through thread cooperation; an example is given in Figure 3-16. In this example, an input array of size 32 is given initially and a block containing 32 threads is assumed. At each successive step, the number of working threads is halved and the strided accessing size is also halved, until single value is obtained. As the order of data elements being processes depends on the block size and array size, the operation (i.e. op) must possess the properties of commutative and associative. An important point to highlight is that both the CUDA block size and the starting array size must be divisible by 2, such that the halving operation results in integer number. Note that the process demonstrated in Figure 3-16 is an example of the divide and conquer strategy with 2 elements in each group. Divide and conquer strategy involves dividing data into groups; each group is reduced to an intermediate results with lesser elements; intermediate results are further divided and reduced iteratively until final results are obtained.

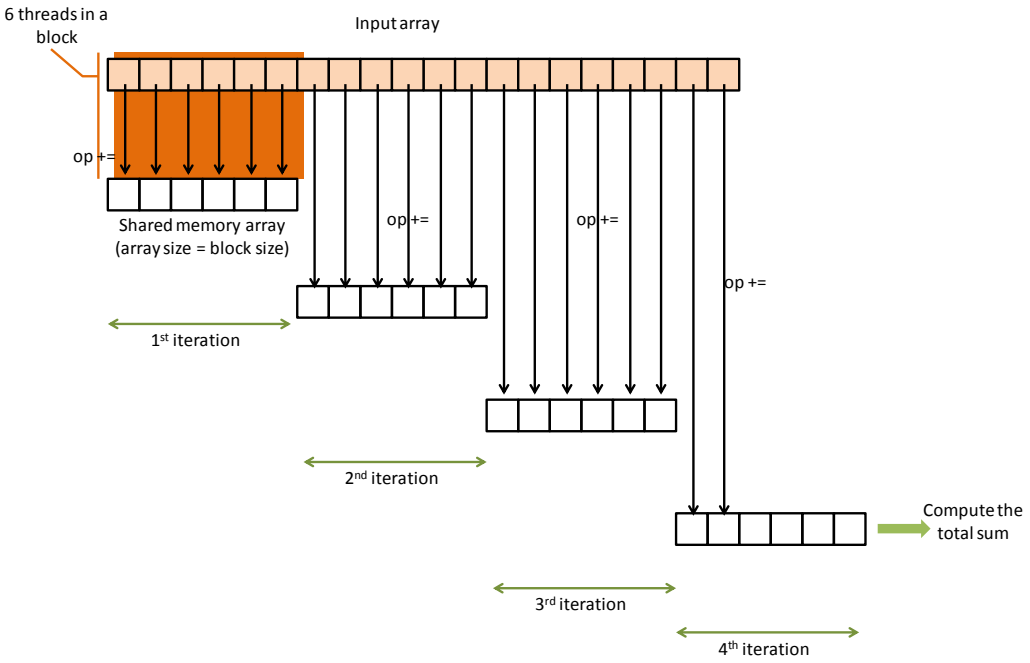

Figure 3-15: An example of using shared memory to support a reduction operation; a small warp containing only 6 threads is used for illustrative purpose.

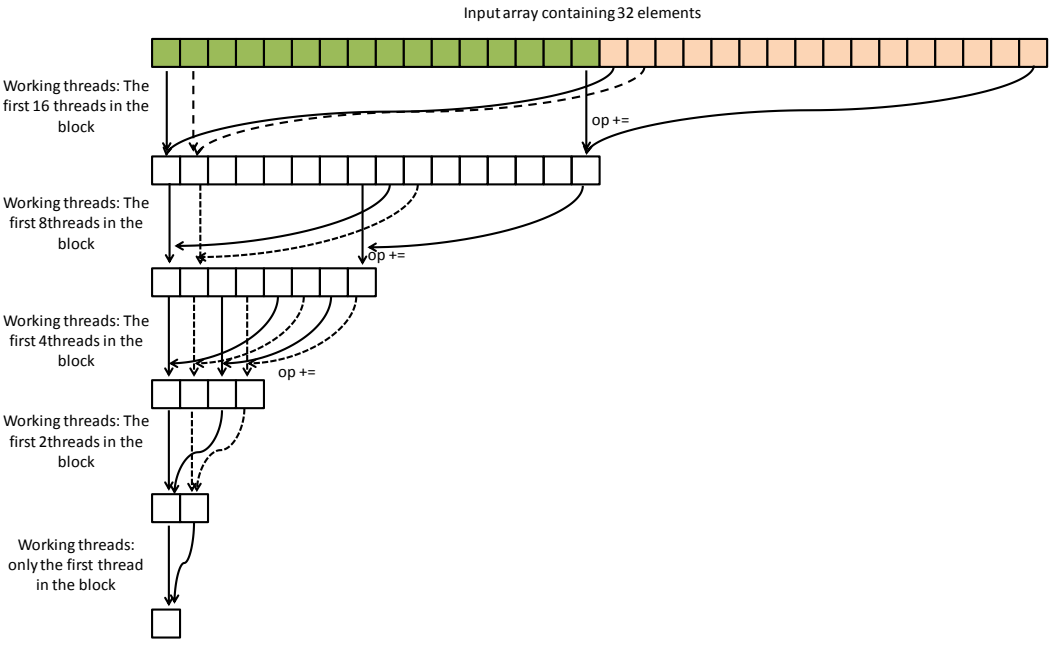

Figure 3-16: An example of reduction operation which calculates the total sum from a 32-element input array residing in either shared or global memory.

Scatter and gather: are essentially the data writing and reading operations.

Stream filtering: involves selection of data subset based on certain criterion, sometimes known as stream compaction; e.g. removing data elements which are negative values, copying unique data elements. It can be regarded as a special class of mapping operation, except that the stream filtering is required to eliminate useless elements from the output array.

Sort: involves alteration of the sequence of input array to generate an ordered output array based on a specific sorting key. For instance, given a sorting key array, K={1, 4, 2, 8, 5, 7}, a value array, V={'a', 'b', 'c', 'd', 'e', 'f'}, a sorting operation transforms V to {'a', 'c', 'b', e', 'f', 'd'} if it is sorted in ascending order. There has been a number of studies focusing on devising efficient GPU-based sorting algorithms, such as GPU sample sort (Leischner et al., 2010), GPU Odd-Even merge sort (Zhang et al., 2011), and GPU quick sort using scan primitives (Sengupta et al., 2007).

The development of efficient CUDA program based on the abovementioned primitives involves finding large amount of data parallelism and effective utilization of GPU resource capabilities. First of all, appropriate computational tasks are identified from the target computing application. The shortlisted computational tasks must satisfy two key criteria, which are massiveness in data parallelism and high computational cost. Essentially, a computational task should be partitioned into many smaller parallel sub-tasks, and each sub-task is further split into even smaller concurrent operations (which is referred as thread-operation in later discussion, for brevity purpose). The task partitioning matches to the two-level hierarchical organization of hardware and software resources, i.e. SPs and SMs, threads and blocks, and thus promises for optimal utilization of GPU resources.

Depending on the nature of computations, the notion of sub-task may refer to a well-defined duty or simply a group of thread-operations. An example of the first case is when a sub-task responsible for constructing a PCA model; a second case example is when a sub-task assigned to one section of a matrix summation operation. Most of the time, the former involves more complex computations than the latter. Apparently, the thread configuration for the second case is more flexible, as the only requirement is to launch sufficient threads regardless of the block size and grid size. On the other hand, when a sub-task is required to execute a more defined and complex duty, the number of threads in each block must be chosen carefully to achieve high degree of parallelism but at the same time avoid unnecessary long waiting time. Based on these considerations, the optimal thread configuration for one computational task may not be the same as another. Therefore it is common to separate tasks which behave distinctly into different CUDA kernels for optimal performance.

3.3.3. **Performance optimization**

Figure 3-17 shows the typical relationship of computational cost and programming effort during the process of transforming a sequential code to GPU-compatible code. A major observation from the plots is that as increasing amount of effort (red curve) spent on optimizing GPU code, the overall computational cost (black curve) is gradually reduced until a plateau is reached. At this stage, programmers are required to investigate the GPU code painstakingly for finding optimization opportunity, or find additional sequential tasks to be ported to GPU despite the inadequacy of the tasks; and yet the resulting improvement in computational efficiency can be extremely little. The region bounded by two dotted vertical green lines represents more practical and realistic performance goals for GPU computing, which guarantees for performance

that is not far from the plateau. In this work, we attempt to arrive at this performance region.

A commonly used performance metric for parallel computing algorithm is the speedup ratio, as given in Equation (3-2), which will be used as the main performance indicator in this study.

$$speedups = \frac{computing\ time\ taken\ by\ sequential\ method}{computing\ time\ taken\ by\ HPC\ method} \qquad (3\text{-}2)$$

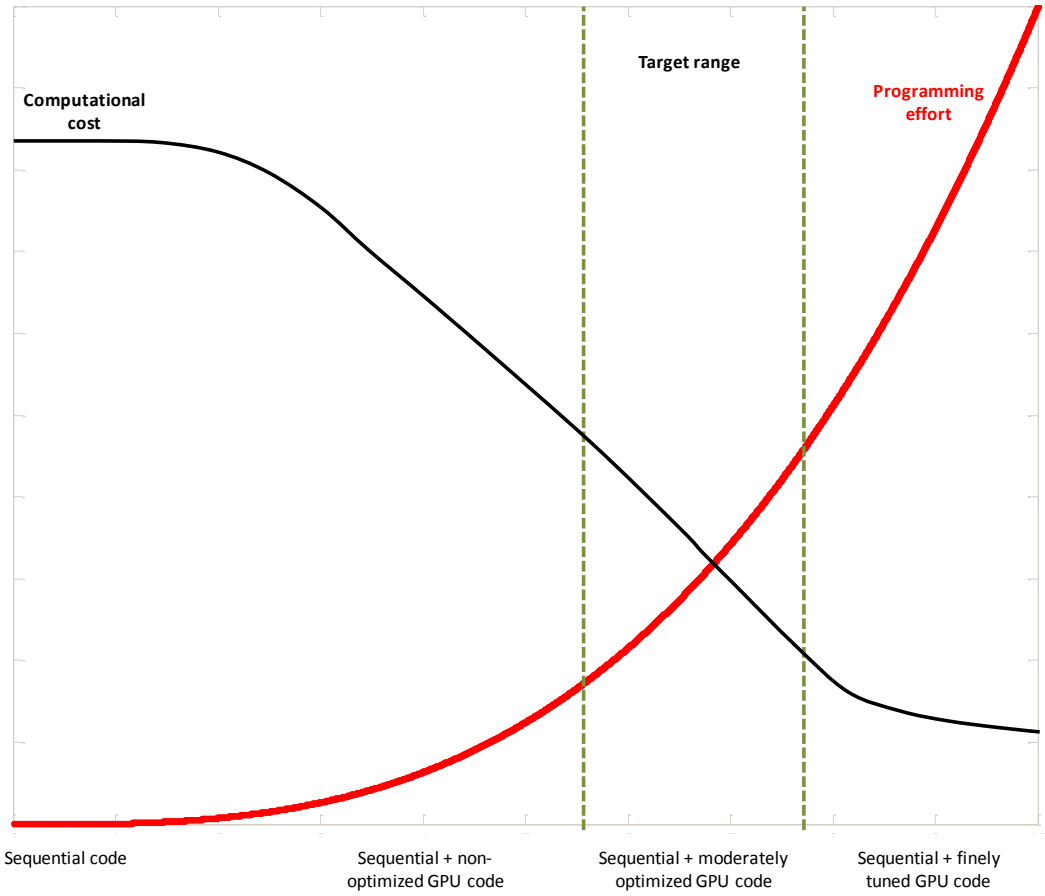

Figure 3-17: Typical (hypothetical) relationship between computational efficiency improvement and programming effort; the green boundary specifies the recommended target range.

Here, we introduce several commonly used strategies for improving CUDA kernel performance.

Data transposition: It exchanges the position of rows and columns, i.e. i^{th} row is rewritten as i^{th} column, j^{th} column becomes j^{th} row. Mathematically, it is expressed as ***A***', or $\boldsymbol{A}^{tr}$, or $\boldsymbol{A}^{t}$, given ***A*** is the original matrix. This approach is useful for global memory access, in which successive threads in a warp attempt to access successive elements in a column. Since successive column elements are stored apart with a gap equal to row width (assuming 32-bit element), warp accessing to them require multiple memory transactions, and thus seriously increases memory latency. By transposing the matrix, much lower memory latency can be achieved.

Data padding: It intentionally includes artificial values like zeros to expand the size of a matrix of an array. The main purpose of data padding is to realize the coalesced memory access to global memory; an illustration example is given in Figure 3-18.

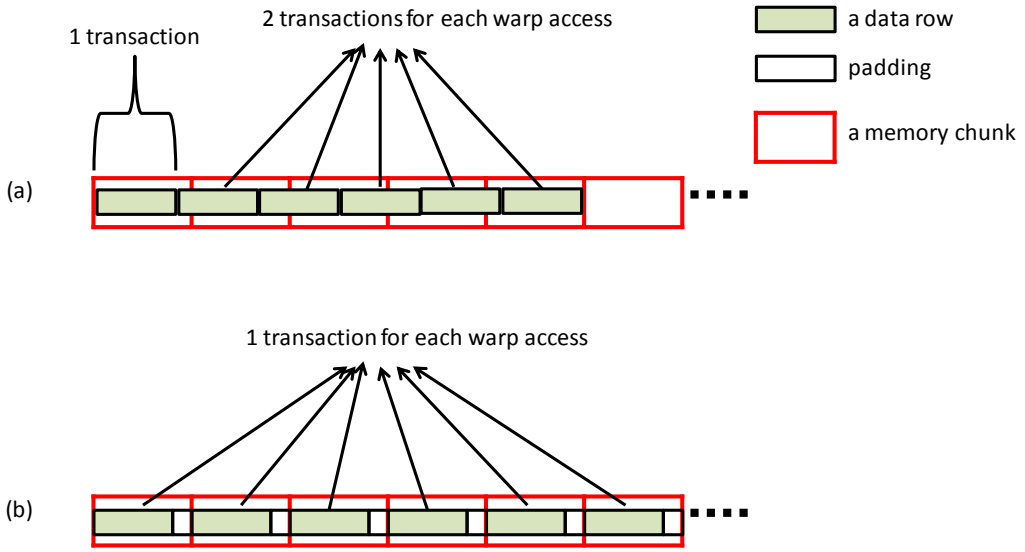

Figure 3-18: An example showing the use of data padding technique to achieve coalesced global memory access, where a data row is accessed by a warp.

Tiling: It loads data to shared memory in tiles, where each tile consists of a number of data elements fitting nicely into the banks of shared memory.

Re-computation: There are situations where intermediate results computed in the preceding kernel are needed in the subsequent kernel, where both kernels belong to the same application. If the writing or reading of intermediate results to or from global memory (which serves as communication platform between kernels) is expensive, re-computations can be more effective.

Experimentally determined optimal thread configuration: In cases where the underlying computing task involves complex computations, it is difficult to identify the optimal thread configuration based on visual investigation of GPU code. Numerical experiments which provide measurements of computational time at various thread configurations, allows for more effective optimization.

Profiling: NVIDIA Visual Profiler (NVIDIA, CUDA TOOLKIT DOCUMENTATION) and NVIDIA Nsight Visual Profiler (NVIDIA, NVIDIA Nsight Visual Studio Edition) are performance profiling tools developed by NVIDIA, which provide critical feedbacks for optimization of CUDA C/C++ applications. Through tracing activities occurring on both the host and GPU, the tools produce useful performance indicators allowing developers to identify performance bottlenecks. Key performance data include application timeline showing time taken for CUDA API calls, memory transfers and CUDA kernel execution, memory requests versus memory transactions, frequency of bank conflicts, achieved memory bandwidth, amount of resident versus available threads/ warp/ block, and etc.

3.4. GPGPU Applications

GPGPU parallel computing is a relatively new technology which has become increasingly popular in academia and research fields. Successful applications have been reported from a wide range of areas, including computer science, engineering, physics, material science, medicine, molecular biology, chemistry, and many more. (Moreland et

al., 2003; Garcia et al., 2008; Sengupta et al., 2007; Krüger et al., 2003; and Manssen et al., 2012) demonstrate the application of GPGPU computing on a number of standard algorithms, including fast Fourier transform (FFT), k-nearest neighbor search, scan primitives, sparse matrix algebra, linear algebra, random number generation, and etc. Remarkable GPGPU performance has also been achieved on specific research areas, such as molecular dynamics simulations (Anderson et al., 2008) and molecular modeling (Stone et al., 2010); medical imaging applications (Heng & Gu, 2005; Shams et al., 2010) and surgical simulation (Taylor et al, 2008); chemical informatics (Haque et al., 2010; Liu et al., 2011; Ma et al., 2011); bio-informatics (Langdon & Harrison, 2008; Liu et al., 2007; Schatz et al., 2007). In particular, GPGPU studies in chemical engineering have so far focused on system modeling and simulations, some representative examples are gas-solid flow simulations (Xiong et al., 2010; Xu, et al., 2012), turbulent flow simulation (Shinn et al., 2010), powder mixing simulations (Radeke et al., 2010), combusting modeling (Xu et al., 2012). In this study, we attempt to explore the use of GPGPU parallel computing technique to a different area, i.e. data mining on chemical process operations. Specifically, we aim to develop efficient GPU-based data mining algorithms for tackling large data problems in chemical industries; our main focus is on the topic of optimization and pattern recognition.

3.4.1. **Optimization**

In this section, we survey a number of GPGPU-based optimization studies. Majority of the works are mainly focused on population-based optimization techniques, such as GA, PSO, ant colony optimization (ACO) and tabu search. An obvious reason is that population-based optimization methods are inherently rich in data parallelism and at the same time they are computationally expensive. Early works like (Fok et al., 2007) and (Li et al., 2007) successfully accelerate evolutionary-based optimization with the use

of shader programming. Since the introduction of CUDA programming in year 2006, there have been an increased number of studies reported on GPU-based optimization; several examples are discussed as follows.

Robilliard et al. (2009) devise an interpreter to handle multiple genetic programming (GP) programs in GPU. The authors propose two schemes of deploying the interpreter. One restricts threads running on the same SM to interpret the same GP program, while the other allows each thread to interpret its own GP program. As could be expected, the former outperforms the latter, with a factor up to 15 times, owing to the absence of branch divergence. In (Zhu et al., 2010), the authors propose a SIMD-tabu search algorithm using CUDA programming. The algorithm is specially designed for the quadratic assignment problem (QAP) which involves the assignment of *n* items to *n* locations while optimizing overall distance. To avoid expensive communication between host and GPU and also among the threads, each thread is responsible for an independent tabu search. During evolution of population, each thread is assigned randomly to one of the 4 operations based on user-specified probabilities; the options are do nothing, mutation with random swap, re-initialization, and replacement with mutated best solution. Satisfactory speedups of 25-40 are obtained on a number of QAPLIB test datasets.

Particle swarm optimization is another popular topic in GPGPU optimization studies. Mussi et al. (2011) present two parallel versions of PSO which are characterized by different usage of global memory. One of the GPU-based PSO restricts a swarm to single CUDA block (termed as SyncPSO), while the other distributes candidates solutions of a swarm to multiple CUDA blocks (termed as RingPSO). As only solutions within the same swarm are required to communicate, the first approach eliminates the use of

global memory by storing the solution data in local registers; while the second approach makes use of multiple kernels for solutions synchronization in between various stages of optimization, and communication is achieved through global memory. It should be apparent that SyncPSO is more appropriate for optimization problems with low dimensionality due to the limitation of local register space. Moreover, sufficiently large number of swarm groups is needed for fully utilization of SMs. Experimental results show that RingPSO outperforms SyncPSO, and thus implying that global memory latency and kernel call overhead are counterbalanced by the advantages of parallelization. In fact, the multi-kernel approach has already been discussed by Veronese et al. (2009) and Zhou et al. (2009), in which 88 and 11 speedups are reported in their works, respectively. These two works use 3 similar CUDA kernels, for fitness value computation, updating of current local and global best position of each particle, updating of velocity and position of each particle. A major algorithmic difference between the two studies is that one employs the MersennTwister CUDA SDK code for random number generation, and the other generates a set of random number on CPU which is copied to GPU global memory before the first generation of optimization.

Ant colony optimization which can be regarded as a special type of PSO also matches to the SIMT model of CUDA-enabled GPU. ACO incorporates the concept of pheromone which is a chemical substance left by ants during food searching, in which pheromone concentration provides the searching direction for ants. As in reality, concentration of pheromone will decrease over time through evaporation, thus there is additional computational task for managing pheromone concentration. In the work (Zhu & Curry, 2009), the authors retain the tasks of pheromone evaporation and update in CPU and overall 200-400 speedups are reported. While the generation of new ant solutions based on the updated pheromone concentration, cost evaluation, and ant

local search are parallelized in GPU. On the other hand, Bai et al. (2009) include all stages of ACO in GPU by using 4 separate CUDA kernels, namely solutions construction, iteration-best choosing, pheromones evaporation, and pheromones deposition, while CPU is only responsible for controlling the iteration process. Effective communication and central-management of ants within a colony is achieved by mapping an ant colony to a CUDA block, and the best values obtained from the optimal solutions of these ant colonies at the end of all iterations serve as the final results. As compared to (Zhu & Curry, 2009), relatively low speedups are obtained, which are about 2.3.

Although remarkable improvement in computational efficiency has been reported in the above works, these results are obtained based on relatively simple test functions. Standard benchmark functions, such as sphere function, step function, Rastrign, Rosenbrock, Schwefel, TSP, QAP are employed in these works, which are characterized by straightforward cost evaluation. Real-world chemical process optimization problems always require much more complex cost functions which may also involves stochasticity. Hence, exploring the performance of GPU-based optimization on more realistic problems is highly desirable.

3.4.2. **Pattern Recognition**

Pattern recognition algorithms offer excessive amount of data parallelism for GPU acceleration, as demonstrated by many successful studies. A number of neural network variants has been effectively transformed to GPU-compatible codes, such as the Probabilistic Neural Network (PNN) (Kostopoulos et al., 2014; Sidiropoulos et al., 2012), Fuzzy Neural Networks (FNN) (Juang et al., 2011; Martínez-Zarzuela et al., 2011), Convolutional Neural Network (CNN) (Cireşan et al., 2011; Strigl et al., 2010), Kohonen's Self Organizing Map (SOM) (Prabhu, 2008). Other GPU-accelerated pattern recognition

methods include Support Vector Machine (SVM) (Catanzaro et al., 2008; Liao et al., 2009), k-Nearest Neighbour (kNN) (Garcia et al., 2010; Kostopoulos et al., 2014; Liang et al., 2009), decision trees (Grahn et al., 2011; Sharp, 2008), and Principal Component Analysis (PCA) (Andrecut, 2009; Funatsu & Kuroki, 2010). In this study, we are particularly interested in 2 groups of algorithms, which are image analysis and clustering.

3.4.2.1. **Image analysis**

Image analysis or processing is fundamentally the reversed operation of image rendering, so GPUs which are originally designed for image rendering purpose are readily adapted to the image processing job. There are comprehensive GPU-based image processing algorithms available in well-established libraries, such as NVIDIA Performance Primitives (NPP) (NVIDIA, NVIDIA CUDA ZONE) and ArrayFire (AccelerEyes). However, it has been seen in many research studies that customized GPU programs are still needed in order to fully exploit GPU computing power for specific applications. In the following discussion, we survey a number of literature studies covering both general and application-specific GPU-based image processing algorithms.

Examples of GPU-based acceleration of basic image processing algorithms are (Asano et al., 2009; Castaño-Díez et al., 2008; Fialka & Čadík, 2006; Park et al., 2011; Zhiyi et al., 2008). Castaño-Díez et al. (2008) show that direct application of standard GPU library functions, like the FFT function from CUFFT, linear algebra functions from CUBLAS, offers better image processing performance as compared to the CPU counterpart. In the work (Park et al., 2011), the authors propose a GPU-based linear feature extraction method which involves piecewise fitting of lines to edges. GPU implementation is achieved by mapping threads to pixels, and each thread traverse

through the neighboring pixels sequentially to identify connected edge chain. As it has been highlighted in the article, there is redundant traversing caused by multiple threads, along the same edge chain. This could be the reason for the relatively low performance speedups, which are 3.2.

Fialka et al. (2006) successfully accelerate the convolution image filtering process using GPU shader programming. The authors employ pixel-level parallelization technique, in which each thread sequentially loops over every element in the mask for filter kernel operations. However, the size of mask is limited by the maximum number of instructions allowable in the shader. In addition, histogram equalization which is usually used for enhancing global contrast of images has also been implemented in GPU (Zhiyi et al., 2008). Given a grayscale image, each thread is responsible for generating a sub-histogram based on a subset of data *S*. As the sub-histograms are stored in shared memory, the block size is bounded at $\frac{available\ shared\ memory\ space}{256}$; where 256 are the number of bins in histogram built from a grayscale image. Subsequently, block-level histogram is obtained by combining all the sub-histograms in shared memory. Lastly, only single block with 256 threads is launched for reducing the block-level histograms to a global histogram, which is used for calculating a new gray distribution of the image.

Next, we discuss GPU implementation of non-standard image processing algorithms, such as k-means clustering and PCA, which are only applicable to specific image applications. As it is mentioned in (Asano et al., 2009), k-means clustering can be used to reduce the number of colors to a smaller groups while maintaining image quality. Given a RGB (or color) image of size *n* by *m*, it is divided into *n* or *m* sub-images which are mapped to equal number of CUDA blocks. Within each block, there are *p* threads running simultaneously on the sub-image pixels, whereby the distances to *k*

cluster centers are computed for R, G, and B components sequentially in each pixel; the RGB pixel values are then added to the k_{min}^{th} element of the arrays $R_{sum}[k_{min}]$, $G_{sum}[k_{min}]$, $B_{sum}[k_{min}]$, $Count[k_{min}]$ in shared memory. These shared arrays can be combined in global memory in later stage for determining the new cluster centers at the end of each iteration.

PCA which is an effective dimensionality reduction tool is commonly used in spectral image analysis. In general, the large data size of spectral images needs to be first reduced to a manageable size, using PCA. In the context of image analysis, GPU-accelerated PCA algorithms are discussed in several works, like (Castaño-Díez et al., 2008) and (Jošth et al., 2012). In (Jošth et al., 2012), the authors focus on the computation of correlation matrix, as it is the most expensive task in PCA. Considering the spectra image as a 2D matrix, in which a column consists of all wavelength information (length=*n*) of a particular pixel, and rows represent different pixels (length=*m*). The computation of the correlation matrix (*n*x*n*) can be regarded as averaging of a stack of *m* number of 2D matrices of size *n*x*n.* Parallelization is achieved by mapping CUDA blocks to components of the correlation matrix. The threads within a block calculate a specific component in different matrices simultaneously. In this work, the eigenvalue and eigenvector calculations are carried out sequentially using single thread. On the other hand, Castaño-Díez et al. (2008) explore the use of GPU computing for enhancing the performance of eigenvalue and eigenvector computations. The authors propose a 2-step numerical procedure which concatenates the power method and Hotelling deflation. Power method is used for determining the highest eigenvalue (and the corresponding eigenvector), and Hotelling deflation produces the residual matrix for finding the next highest eigenvalue, in the following iteration. Since linear

algebra operations are the key components, straightforward GPU implementation with CUBLAS library results in speedups up to 10.

However, it has been noticed that applications of these highly efficient GPU-based image processing techniques are mainly limited to medical and surgery applications. For instance, image registration for patient positioning in radiation therapy (Gu et al., 2010; Khamene et al., 2006), image segmentation (Pan et al., 2008), image reconstruction (Cui et al., 2011; Schiwietz et al., 2006). In this study, we extend the use of GPU-based image processing techniques to effective information discovery in chemical process operations.

3.4.2.2. **Clustering**

Distance computations required in clustering algorithms, in the form of object-to-object or object-to-centroid distance, generate enormous amount of data parallelism. Several studies attempt to parallelize the object-to-centre distance computations during k-means clustering, e.g. (Bai et al., 2009, Farivar et al., 2008; Li et al., 2010; Takizawa & Kobayashi, 2006). Parallelization is achieved by assigning 1 CUDA thread to 1 data object; each thread is responsible for calculating distance from the data to all cluster centers so that the nearest centre can be identified. The parallel k-means algorithm proposed in (Takizawa & Kobayashi, 2006) intends to tackle a large-scale clustering problem. It is achieved by distributing data into multiple computing nodes (CPUs), each node comprises of a GPU. Given the limited data parallelism in updating cluster centres, some authors like Farivar et al. (2008) and Takizawa et al. (2006) choose to retain the task in CPU; while Bai et al. (2009) propose to sort the cluster labels in CPU first which facilitates the simultaneous updating of *k* clusters by *k* threads; Li et al. (2010) compute the new centroids using divide and conquer strategy. Li et al. (2010)

present 2 GPU-based k-means algorithms which are targeted at problems with low and high dimensionality, respectively. To address a low dimensional problem, the authors suggest using fast local register for storing object data. Whereas for a high dimensional problem, data object is loaded into shared memory and distance computation is carried out as a matrix multiplication process.

Besides k-means, other GPU-based clustering algorithms include density-based clustering, bird flocking algorithm, and GMM. In (Böhm et al., 2009), the authors define a concept called 'chain' to allow for massive parallelism in DBSCAN method. Multiple chains which are assigned with different starting data points are assigned to different SMs. Within each chain, one core object is considered at a time so that many threads can be created for processing its potential neighbors simultaneously. In order to avoid collision, i.e. chaining of the same data by multiple chains, a collision matrix is maintained for keeping all necessary information. Cui et al. (2011) develop a GPU-based bird flocking algorithm which is significantly more efficient than the CPU counterpart. Bird flocking algorithm moves individual data (i.e. bird) according to the movement of others in its neighborhood, thus requiring object-to-object distance computations. The GPU algorithm consists of 2 CUDA kernels; one assigns N^2 threads to parallelize the N^2 pair-wise distance computations, where N is the total number of birds; the other kernel launches N threads for updating the position and velocity of the N birds simultaneously. Note that the initial positions of birds are randomly generated, and through the flocking process the birds will ultimately organize themselves into separate clusters. In the work (Machlica et al., 2011), a GMM clustering algorithm based on Expectation-Maximization (EM) technique is transformed to a GPU-compatible code. Major data parallelization is realized on the computation of log-likelihood and component's posterior probability, which are independent among the data objects.

To address the issue of fast and infinite growing of data size and data generation speed, efficiency improvement on scalable clustering algorithms are extremely desirable. In contrast to traditional clustering algorithms, scalable methods allow for accurate clustering at unbounded problem size. Wu et al. (2009) propose to process data by batches in order to tackle problem size that is larger than the size of GPU global memory. In each data batch, ordinary k-means is executed and sufficient statistics of the resulting *k* clusters are passed to the following batch. However, this approach incurs an extremely high data transferring cost, as multiple data batches need to be moved from CPU to GPU at every iteration. The authors alleviate the data transferring cost with the use of CUDA stream API, which enables asynchronous memory transfer and streaming among different data blocks. Wasif et al. (2011) employ a similar approach but using a multinode-multiple GPUs computing system. In the work, data batches are distributed to the nodes (CPUs) from which the data is further divided and sent to multiple GPUs. The output of cluster centroids follows the opposite direction of data distribution, where the partial centroids computed by individual GPU are sent back to the parent node, centroids collected from multiple GPUs in a node are then transferred back to the master node from which global centroids are calculated. The global centroids will be broadcasted to all nodes and GPUs at the next iteration, if the termination criterion is not met. In (Cao et al., 2006), the proposed GPU-based method also focuses on a section of data at a time, based on the concept of landmark window and moving window. A major concern of these 3 studies, (Cao et al., 2006; Wasif & Narayanan, 2011; Wu et al., 2009), is that clustering quality of the proposed GPU methods are not validated.

A number of GPGPU studies have also been done on improving the efficiency of incremental clustering algorithms. For instance, Chen et al. (2012) execute an

incremental GMM algorithm in GPU. At each incremental step, a data batch is copied to GPU for the execution of standard EM algorithm and order (i.e. number of cluster) identification. The Gaussian mixture model keeps evolving by merging the new model obtained from current data batch with the historical model into single model, where merging is done by finding statistically comparable components. Chen et al. (2013) and Papenhausen et al. (2013) explore the use of GPU computing on micro-cluster-based incremental clustering methods. Chen et al. (2013) defines cluster granularity based on a mean shift algorithm, whereas Papenhausen et al. (2013) pre-specify a radius threshold. It is also mentioned in (Papenhausen et al., 2013) that data parallelization is achieved by mapping CUDA threads to data object during distance computation and cluster centers updating.

Dong et al. (2013) present an accelerated version of BIRCH algorithm using GPU computing, termed as GBIRCH. BIRCH is a widely used incremental clustering method from which the idea of micro-clusters is originated. GBIRCH demonstrates the use of dynamic parallelism capability offered by NVIDIA's latest Kepler Compute Architecture GPUs. CUDA Dynamic Parallelism feature allows calling of kernel functions (i.e. slave kernels) from another kernel function (i.e. master kernel), and thus enhancing the flexibility and simplicity of GPU programming. In GBIRCH, the entire BIRCH algorithm is executed in GPU, whereby each slave kernel will receive a subset of data from the master kernel. The distance computation and nearest cluster identification are carried out simultaneously in all slave kernels. In cases where a data cannot be absorbed into existing cluster, the data will be returned to the master kernel. Another refinery kernel will then be activated to handle the returned data sequentially in GPU, so as to ensure the constructed CF-tree is accurate by avoiding race conditions. However, GBIRCH is

mainly tested on low dimensional problems. Moreover, as dimensionality increases the performance is also seen to degrade slowly.

Chapter 4. Hybrid CPU-GPU Sim-Opt for Continuous Optimization

Simulation-optimization has gained increasing popularity in chemical industries, particularly in the areas of process design and operation, supply chain management, and sustainability development. Sim-Opt outperform conventional optimization techniques due to the capabilities of handling complex, highly dimensional, and stochastic processes, through incorporating the simulation feature. Although there have been a number of methods proposed for reducing model complexity and thus the simulation time, e.g. dimensionality reduction or model structure simplification based on assumptions. Such approaches are either still required to deal with the originally large input data, as in the case study to be discussed in Section 4.4 or they might incur in a loss of modelling details by using oversimplified process model. Therefore, the tremendously huge computational effort and time required by Sim-Opt is hardly avoidable and causing infeasibility and intractability in many cases. It becomes even more challenging if Sim-Opt algorithm is applied to continuous optimization problems, as the Sim-Opt routine needs to be performed repeatedly and rapidly. To the best of our knowledge, there is no related work found in the literature.

To overcome the long simulation time and to enhance the practicality of Sim-Opt techniques, a hybrid CPU-GPU Sim-Opt systematic framework is proposed in this work. While the existing GPU-based optimization methods are limited to simple and deterministic objective functions as reported in Section 2.5, we provide a useful and efficient optimization solution to real-world chemical problems. Our main contribution in this specific area, i.e. GPU-based optimization, is to explore the use of GPU parallel

Related publications:

A Graphic Processing Unit (GPU) Algorithm for Improved Variable Selection in Multivariate Process Monitoring, Computer Aided Chemical Engineering, 31, 2012;

A Graphic Processing Unit (GPU) Algorithm for Improved Variable Selection in Multivariate Process Monitoring, AIChE Annual Meeting 2012, Pittsburgh, PA.

computing on complex optimization applications wherein stochasticity and simulation model are needed in place of straightforward mathematical equations. Moreover, a comprehensive investigation of important Sim-Opt techniques is also performed in order to identify potential computational tasks or operations to be ported to GPU. The remainder of this chapter is organized as follows. In Section 4.2, we first analyze the nature of various Sim-Opt techniques, and then identify the type of data parallelism involved. The major steps of the proposed framework are provided in Section 4.2; while application of the framework is illustrated using a variable selection monitoring problem in Section 4.3. Lastly, a case study of Tennessee Eastman challenge problem is carried out to validate the performance of the developed hybrid algorithm.

4.1. Structural Analysis of Sim-Opt Techniques

The nature of problem and the type of technique used in the 3 modules of Sim-Opt govern the amount of data parallelism, and thus opportunity for acceleration using GPU. As has already been mentioned, simulation is the most time consuming module in Sim-Opt. The total simulation time in sequential approach is equal to the simulation time of single run multiplies by the total number of simulation runs. The number of simulation runs required in each optimization cycle is determined by both the optimization technique and the degree of uncertainty involved. In the following discussion, we focus on two classes of optimization techniques, population and gradient-based methods, due to the large amount of simulations involved.

Population-based methods like GA, PSO and simulated annealing (SA) produce *P* candidate solutions, where *P* is a user-specified population size. Assuming there is zero uncertainty, one simulation is required for fitness measurement of one candidate solution. Hence, *P* simulation runs will be needed in each optimization cycle. On the

other hand, in order to find the optimal search direction in gradient-based optimization problems, it requires performing O(*N*) simulations, where *N* is equal to the number of input variables. Although there are enhanced versions of gradient-based methods such as LR, PA, frequency domain method (FDM) and RSM which requires reduced amount of simulation runs, their applicability are limited (refer to Section 2.5.1.3). Furthermore, in cases where uncertainty needs to be accounted for, additional *M* simulation runs (as seen in Figure 2-1) representing different simulation paths will be needed for fitness measurement of single candidate solution; this special type of optimization solution is sometimes referred as stochastic optimization.

In order to characterize data parallelism in Sim-Opt techniques, we treat each module as a separate task for identification of parallelism, and also study the relationship between neighboring modules. At each optimization cycle, gradient-based methods generate only one candidate solution, thus there is no apparent data-parallelism. Although population-based methods generate *P* candidate solutions in each cycle, the off-spring generation process is intrinsically sequential due to the stochastic operations involved e.g. mutation, crossover. It is worth nothing that off-springs are generated from best few parents and this requires sorting of the parents based on fitness values. Such sorting can be computationally demanding when *P* is large and multiple objective values are involved (i.e. multi-objective optimization). GPU-based sorting could be helpful if the sorting has become the computational bottleneck. On the contrary, the tasks of simulation and objective evaluation offer massive amount of data parallelism. As the system response of all *P* candidate solutions (or *M* simulation paths) are determined based on the same simulation model, the *P* (or *M*) simulations can be carried out simultaneously. Similarly, in objective evaluation, the fitness values associated with the *P* candidate solutions (or *M* simulation paths) can be computed

concurrently, whereby same objective function is used with different input values (i.e. system response). A further point is that the *M* fitness values will be subsequently reduced to single value which indicates the fitness of a candidate solution.

Based on the previous discussion, we can establish the inter-module relationship by looking at each neighboring pair of modules. Firstly, as the fitness of a candidate solution is represented by one objective value, it is a one-to-one relation between objective evaluation and candidate solution. Moreover, the relation between the number of candidate solution to simulation runs can be many-to-many (population-based method), one-to-one (LR and PA) or one-to-many (finite differences, RSM and stochastic optimization); whereas simulation-to-objective evaluation relation is always one-to-one. The type of inter-module relationship indicates if the two neighboring modules can be combined into single CUDA kernel, assuming both modules are GPU-parallelizable. The various types of data parallelism found in different Sim-Opt techniques are summarized in Figure 4-1.

In this chapter, a GPU solution is developed for a population-based stochastic Sim-Opt problem where massive amount of data parallelism is available due to the structure of GA search paradigm. A similar GPU solution should also be possible for gradient-based Sim-Opt problem, except that the attainable amount of computational acceleration is strictly constrained by the problem dimensionality and the degree of uncertainty. Previous works like Arora et al., 2010 and Statz et al. 2013 have shown that simultaneous computation of sensitivity calculations, which are the most computational demanding operations in gradient-based techniques, on GPU offers attractive amount of speed improvement. In order to apply such GPU-parallelized gradient-based

optimization solution to Sim-Opt problems, the simulation model needs to be implementable within GPU.

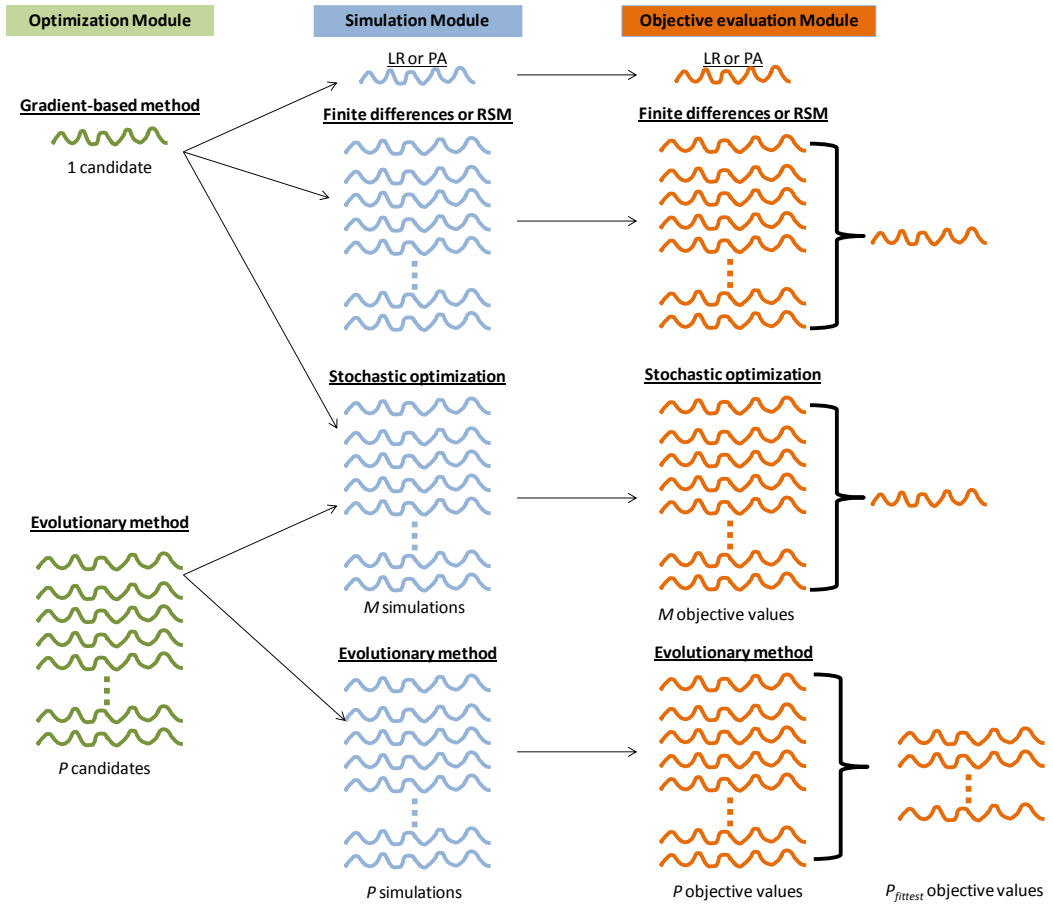

Figure 4-1: Notion of data parallelism in each module of Sim-Opt; independent operations are denoted by same colored curves.

4.2. Systematic Procedure for Developing Hybrid CPU-GPU Sim-Opt Algorithm

A systematic approach for formulating an efficient hybrid CPU-GPU Sim-Opt algorithm is detailed in this section. Major steps include computing time examination, GPU task selection, data transfer consideration, CUDA kernel design, and CUDA kernel optimization. Individual steps are described as follows.

Computing time examination: Since GPU programming is not an easy job, it is desirable to prioritize the most expensive tasks. Computing time consumed by different portions

of the algorithm can either be determined through numerical experiments (if sequential code is available) or based on theoretical complexity analysis of the algorithm.

GPU task selection: The GPU tasks are selected based on three criteria, namely high computational cost, rich data parallelism, and GPU-compatible. Note that the computational cost has been determined in the previous step, and the amount of data parallelism can be estimated by the type of optimization technique used, as discussed in Section 4.1. The last criterion, GPU-compatibility, will be satisfied if the computations involved in the task are supported by GPU, specifically CUDA-enabled GPU. There are cases where the simulation model is extremely complex like those built on high-end simulators like MathWorks's SIMULINK, implementing of such simulation on GPU could be tremendously taxing.

Data transfer consideration: After selecting suitable tasks for GPU parallelization, next step is to assess the data transferring requirement, in terms of frequency and data size. It should be noticed that, input data to a module is generated from the preceding module, in a Sim-Opt problem. Hence, there are 4 key input data which include simulation paths, candidate solutions, system responses, and objective values. If same set of simulation paths is used throughout all optimization cycles, the data only needs to be transferred once from CPU to GPU. On the contrary, the other 3 input data are required to be transferred at every cycle. The size of these input data depends on the number of uncertain variables, problem dimensionality, and number of objective values.

CUDA kernel design: As mentioned in Chapter 3, CUDA programming is essentially a two-level partitioning whereby a task is divided into subtasks and then thread-operations. Based on the structural analysis of Sim-Opt as summarized in Figure 4-1, there are two potential GPU tasks, namely simulation and objective evaluation. A

subtask represents a simulation run (or a fitness measurement) in simulation module (or objective evaluation module). The use of fast memory and cooperation work within a subtask are dependent on the mathematical model of simulation (or objective function).

CUDA kernel optimization: At this stage, applicability of optimization techniques as presented in Section 3.3.3 should be examined. Besides, combining two tasks into single kernel has the advantages of reducing kernel call overhead time and memory copying time, especially when intermediate results are stored in shared memory or local memory. Based on the inter-module relationship, two neighboring modules or tasks which comprise of same amount of subtasks, e.g. *P*-to-*P* and *M*-to-*M*, can thus be combined. However, if individual tasks achieve best performance at different thread configurations, it is still advisable to keep them in separate CUDA kernels.

4.3. GPU Application to the Variable Selection Problem

Implementation of the proposed systematic procedure for developing a hybrid CPU-GPU Sim-Opt algorithm is demonstrated on a variable selection monitoring problem. A PCA model for process monitoring can be constructed with all the measured process variables or a subset of these measured variables. Here, we are interested to identify the subset of variables which is sensitive to faults under certain system behavior. Therefore a PCA model constructed using this subset, termed as reduced PCA model, is expected to have improved monitoring performance. We trust that when the process behavior changes with time, the subset of variables selected could be different, and thus the variable selection routine needs to be repeated in order to provide an adequate process monitoring. As a result, it is important to have an efficient variable selection algorithm to avoid any delay or use of outdated monitoring model. To identify these variables, Ghosh et al. (2014) proposed a multi objective GA-based stochastic

optimization technique, termed as GA-PCA algorithm. This algorithm seeks to minimize the cumulative error rate, defined as sum of missed detection and false alarm rates, while at the same time minimizing the number of selected variables; where an evolutionary optimization method GA is used. The improved monitoring performance, in terms of the cumulative error rate, achieved by reduced PCA model is shown in Figure 4-2. From the perspective of Sim-Opt, the reduced PCA model serves as the simulation model in which component scores represent the system responses. Once the system response of testing data has been determined, it will be classified as normal or faulty data which is used for objective value computation. Note that both model parameters (i.e. eigenvalues and loadings) and model structure (i.e. number of principal component selected) are varied at every optimization cycle.

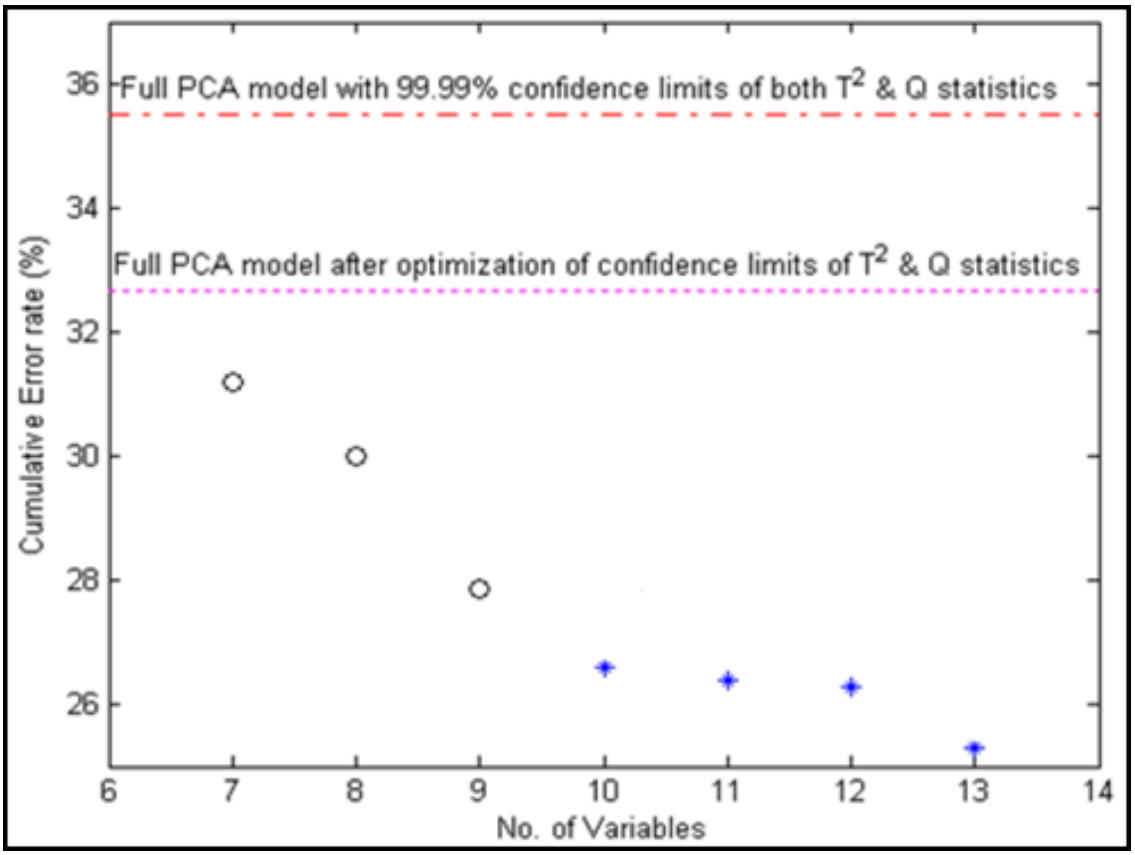

Figure 4-2: Comparison of monitoring performance between full PCA models and reduced PCA model[7].

4.3.1. GA-PCA algorithm

Given a training dataset (i.e. normal operation data) and a testing dataset (i.e. normal and faulty operation data), our goal is to identify the best subset of variables satisfying below objective function,

$$\min_{x \in R}(false\ alarm\ rate + miss\ detection\ rate + no.of\ selected\ variables) \quad (4\text{-}1)$$

GA encodes the variable selection in a binary vector with *N* elements, where 1 denotes a particular variable is selected and 0 means otherwise. A reduced set of training data obtained by removing those unselected variables is used for PCA modeling. Essentially, PCA transforms the originally correlated variables into a set of uncorrelated variables, from which *q* latent variables are selected to account for the user-specified data variability; 95% is used in this work. Mathematical formula of PCA is given below,

$$X = TP^T + E \quad (4\text{-}2)$$

where, *X* represents the original input data, *T* denotes the principal component scores, *P* denotes the principal component loadings, and *E* refers to the residual noise not covered by the *q* latent variables; as *T* and *P* only account for the first *q* variables.

During objective evaluation, testing data is classified as normal or faulty class by comparing the Hotelling's T^2 and Q statistics with the control limits T^2_α and Q_α, which are obtained from the training dataset. As stated in (Kourti & MacGregor, 1995), T^2 and *Q* statistics are computed from PCA score values, as follows,

[7] Courtesy: (Ghosh et al., 2014)

$$T^2 = \sum_{i=1}^{q} \frac{(t_i)^2}{\lambda_i} \quad (4\text{-}3)$$

$$Q = \sum_{i=q+1}^{V} (t_i)^2 \quad (4\text{-}4)$$

where, t_i denotes the score value of principal component i, λ_i represents the eigenvalue of principal component i, q is the number of principal components chosen, and V is the total number of process variables. The score value t_i is a dot product computed from a test sample and an eigenvector. Based on the classification results, an overall objective value can be computed according to Equation (4-1); where false alarm rate is defined as the percentage of normal data which are found violating both the upper control limits, whereas missed detection rate is defined as the percentage of faulty data which are found below the two limits. In each GA generation, P objective values will be computed to indicate the fitness of the P candidate solutions i.e. variable subsets.

To solve this multi-objective optimization problem, the non-dominated sorting genetic algorithm NSGA-II is employed here. NSGA is initially proposed in (Goldberg, 1989) and later modified by Deb et al. (2002) as NSGA-II. There are two key values used for the sorting operation in NSGA-II, namely front assignment and crowding distance. Chromosomes that are superior in at least one objective value than others are assigned with front 1. Front 2 assignments are reserved for chromosomes that are only dominated by those at front 1, and subsequent fronts are assigned in similar way. Within each front, the crowding distance of a member j is computed as below,

$$d_{crowding,j} = \sum_{i}^{i=L} \frac{Obj_{next,i} - Obj_{prev,i}}{Obj_{max,i} - Obj_{min,i}} \quad (4\text{-}5)$$

where, L denotes the total number of objectives (L=2 in this problem); $Obj_{max,i}$ and $Obj_{min,i}$ are the maximum and minimum value of objective variable i, respectively; $Obj_{next,i}$ and $Obj_{prev,i}$ refer to the next higher objective and previously lower value to j, respectively, which are obtained from a sorted list of chromosomes. The candidates of current and previous generations are combined and sorted using NSGA-II, from which the best P' candidates are selected as the parent seeds to generate new off-springs for next generation, through mutation and crossover operations. The 3 modules, PCA modeling, objective evaluation and optimization, will be repeated until a pre-specified number of generations is reached.

4.3.2. **GPU implementation**

In this section, we develop a hybrid CPU-GPU-GA-PCA algorithm using the proposed systematic approach. The major steps are discussed as follows.

Computing time examination: Numerical experiments have been performed with different population and generation sizes in order to identify the most computationally demanding tasks. As it can be seen from the top part of Figure 4-3, at moderate population and generation size, PCA modeling and objective evaluation are the computational bottleneck which collectively account for 85% of total time. When larger population and generation size are used, non-dominated sorting has become dominant by consuming 66% of the total time. Hence, in GA-PCA algorithm, there are three computationally expensive tasks, namely PCA modeling, objective evaluation and non-dominated sorting.

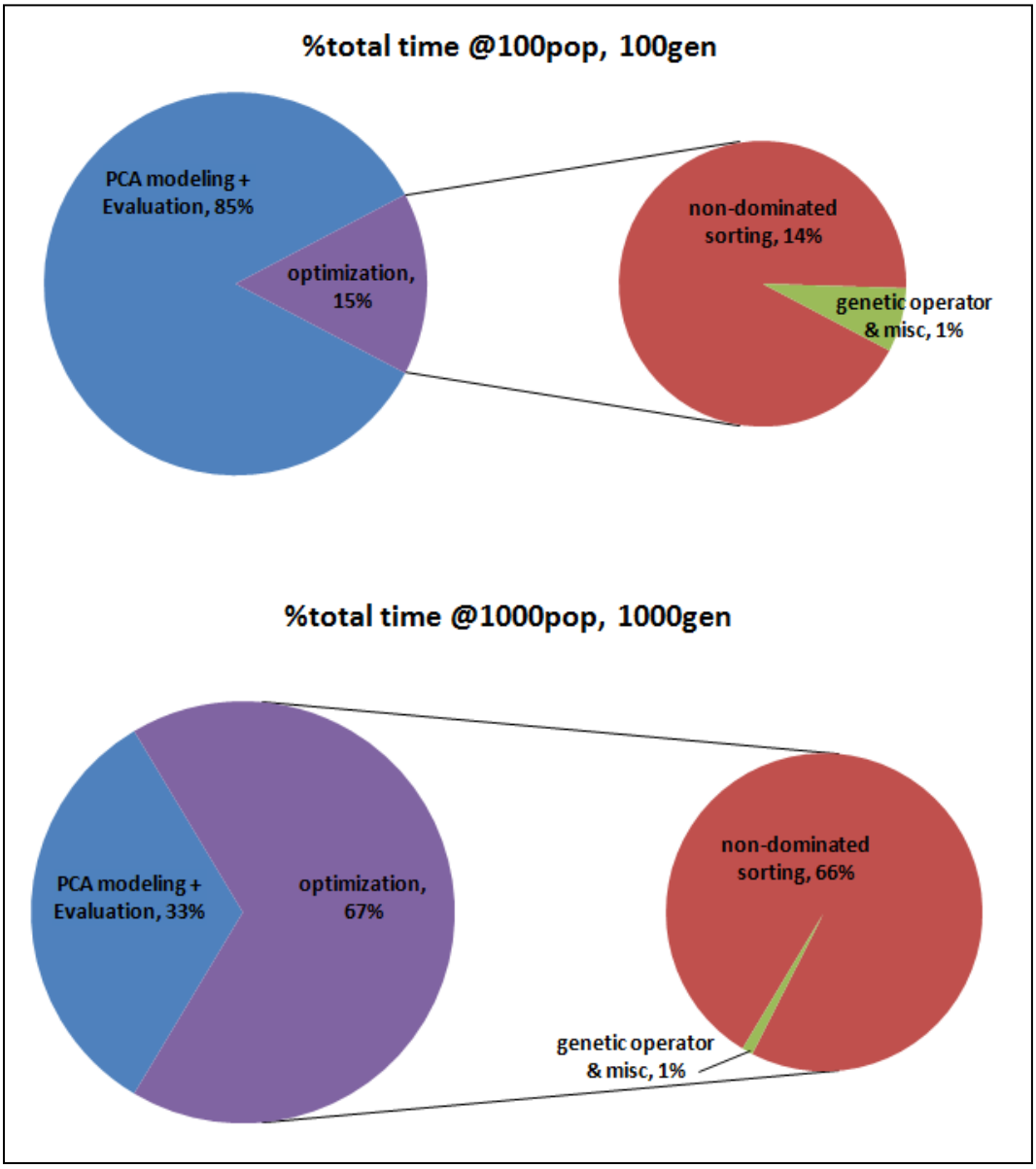

Figure 4-3: Computing time distribution in GA-PCA algorithm; top figure shows the results obtained at moderate population and generation size; bottom figure is based on larger population and generation size.

GPU task selection: We turn now to examine the 3 identified tasks, PCA modeling, objective evaluation and non-dominated sorting, for the 3 criteria. The first requirement of having high computational cost has already been verified with the numerical experimental results. Typical GA optimization problems employ large *P* population size to guarantee for global optimal solution, so PCA modeling and objective evaluation should contain sufficient amount of data parallelism. Furthermore, the availability of

numerical solutions of PCA supports for GPU programming. In this work, we employ the Jacobi transformation numerical PCA algorithm as given in (Flannery et al., 1992); where the underlying concept is to apply a series of plane rotations so as to achieve zeroing for the off-diagonal elements. Since objective evaluation involves simple operations like comparison (data classification), summation and division (error rate computation), GPU programming of this task is straightforward.

Next, we examine the NSGA-II task for data parallelism and GPU-compatibility. First of all, the dominating relation among the *P* chromosomes is determined through performing $(P^2 \times L)$ comparison operations. As the comparison operations are independent to each other, the amount of data parallelism is equal to $(P^2 \times L)$. Supposing the dominating relation is stored in a *P* by *P* matrix ***D***, identification of chromosomes at each front require examining P_{left} rows in ***D***, where P_{left} denotes the number of chromosomes that have not been assigned with front number. Although the computations of crowding distance at different fronts are not parallelizable, distance computations of all members in the same front can be executed simultaneously. Furthermore, NSGA-II requires simple computations like comparison and distance computation, so parallelizing it on GPU should offers promising improvement in computational speed.

Data transfer consideration: To construct a PCA model based on a subset of variables, it needs a training dataset and a binary vector called chromosome which specifies the selection of variables. For the entire population *P*, the same training dataset along with *P* chromosomes are used to build the *P* PCA models. Although both training data and chromosomes are read-only data, they are stored in global memory because they are usually too large for CUDA constant memory, and also do not have apparent

broadcasting characteristic. In regards to the objective evaluation task, it requires a testing dataset along with the PCA model details (eigenvalues, eigenvectors, Q_α and T^2_α). Like training data, the testing data is also stored in global memory for similar reasons. As eigenvalues and eigenvectors which define a PCA model are the output data from another GPU task, so they also reside in global memory. Likewise, NSGA-II sorting task directly makes use of the objective values which is written in global memory by the preceding GPU task. On the whole, constant memory and texture memory are not used as the special memory accessing patterns, i.e. broadcasting or spatial locality, are not needed for the 3 tasks.

CUDA kernel design: For the two tasks, PCA modeling and objective evaluation, an intuitive choice for the first level task partitioning is to regard one candidate as one subtask, so there are *P* subtasks. Within a PCA modeling subtask, all CUDA threads work cooperatively to perform the Jacobi transformation and control limits computation. As the covariance matrix which is used for Jacobi transformation are essentially the dot products between each pair of variables, multiple CUDA threads can be used to parallelize the multiplication operations in a particular dot product. Note that thread-operation parallelization is realized in single element (i.e. dot product) of covariance matrix. Apart from that, limited data parallelism is achievable from the remaining operations in Jacobi transformation and control limit computations, these include summation of the diagonal elements with reduction operation in shared memory, concurrent swapping of several elements during plane rotation, and choosing of largest eigenvalue using divide and conquer strategy. During Jacobi transformation, multiple reading and writing to variables chromosomes, eigenvalues, and eigenvectors are needed, so these variables are copied to shared memory for fast memory accessing. On the other hand, a subtask in objective evaluation is much richer in data parallelism due

to the independent operations involved among the testing samples. These operations include score value computation and sample classification. It should be noted that CUDA threads are mapped to testing samples instead to process variables, as the number of testing sample is typically much larger than the number of process variable.

First of all, it should be obvious that standard GPU-based sorting algorithms such as GPU sample sort (Leischner et al., 2010), GPU Odd-Even merge sort (Zhang et al., 2011), GPU quick sort using scan primitives (Sengupta et al., 2007) and GPU sorting functions available in Thrust library (Thrust), are not applicable for this case due to the presence of multiple objectives. Considering the iterative nature of NSGA-II in determining successive fronts, we split the task into three separate CUDA kernels. As has been mentioned early, front assignment for front i can only be carried out after front i-1, thus the same assignment process is repeated by incrementing the front number until there is no unassigned chromosome. The first CUDA kernel involves calculating the dominating relation matrix $\boldsymbol{D}_{pxp}$; in which if chromosome i is dominated by chromosome j then the element ij is assigned with a value of one, otherwise it has value zero. A vector d_p of length equal to P is used store the number of dominating chromosomes, for instance, if i^{th} element of d_p has value 1, it indicates i^{th} chromosome is dominated by one other chromosome. The values of d_p are also computed in the first CUDA kernel, by mapping P blocks to P rows and T threads to P columns of $\boldsymbol{D}_{pxp}$. Note that T can be larger or smaller than P, but it must ensure that there is no more or less than P variables being accessed to avoid incorrect computation.

Once the first kernel is done, it enters the iterative process of repeating two consecutive CUDA kernels; one is for identifying chromosomes that are not dominated by any other member and the other one is for computing the crowding distance, in a particular front.

In the front assignment kernel, P blocks are assigned to P chromosomes wherein single thread is used to check if the associated element in d_p is zero. After chromosomes at the current front have been identified, d_p is updated so that those at higher front will become visible. In regards to a CUDA block b, multiple threads are used to determine the number chromosomes (says u) which are dominating b^{th} chromosome and also at the current front, by simultaneously examining $\boldsymbol{D}_{pxp}$ and d_p, then one thread is used to decrement the b^{th} element in d_p by u. For effective parallelization of crowding distance computation, the objective values of all members at current front are copied to 2 dummy vectors (as L=2), each of them is sorted using the sorting function in Thrust library. Due to the simplicity of computation involved, as seen in Equation (4-5), there is essentially one level of parallelism. In this kernel, the amount of data parallelism is equal to the number of members at a particular front.

CUDA kernel optimization: Although the two tasks, PCA modeling and objective evaluation, are divided into same number of P subtasks, combining them into single kernel shows poorer performance. This is mainly because they are characterized by different amount of data parallelism, thus requiring different block sizes for optimal performance. For instance, a large CUDA block used in PCA modeling will only take up the resources without doing useful work i.e. idling; whereas having too little CUDA threads in objective evaluation kernel will lead to under-utilization of GPU resources.

In order to achieve efficient memory access, data transposition technique is deployed at both PCA modeling and objective evaluation kernels. During the computation of covariance matrix in PCA modeling kernel, consecutive threads access the same variable of different training samples. Hence, the training data is transposed such that data corresponding to the same variable is allocated in successive memory addresses for

coalesced accessing, as illustrated in Figure 4-4. In conventional eigenvalue decomposition algorithm, eigenvectors are stored as column-vectors as seen in Figure 4-5. For illustration purpose, assuming the number of variables *V* is equal to the number of banks in CUDA shared memory. During computation of score values, elements of a particular eigenvector are accessed by consecutive threads in shared memory, so it is advisable to allocate the elements in different banks for prevention of bank conflicts, as depicted in Figure 4-6.

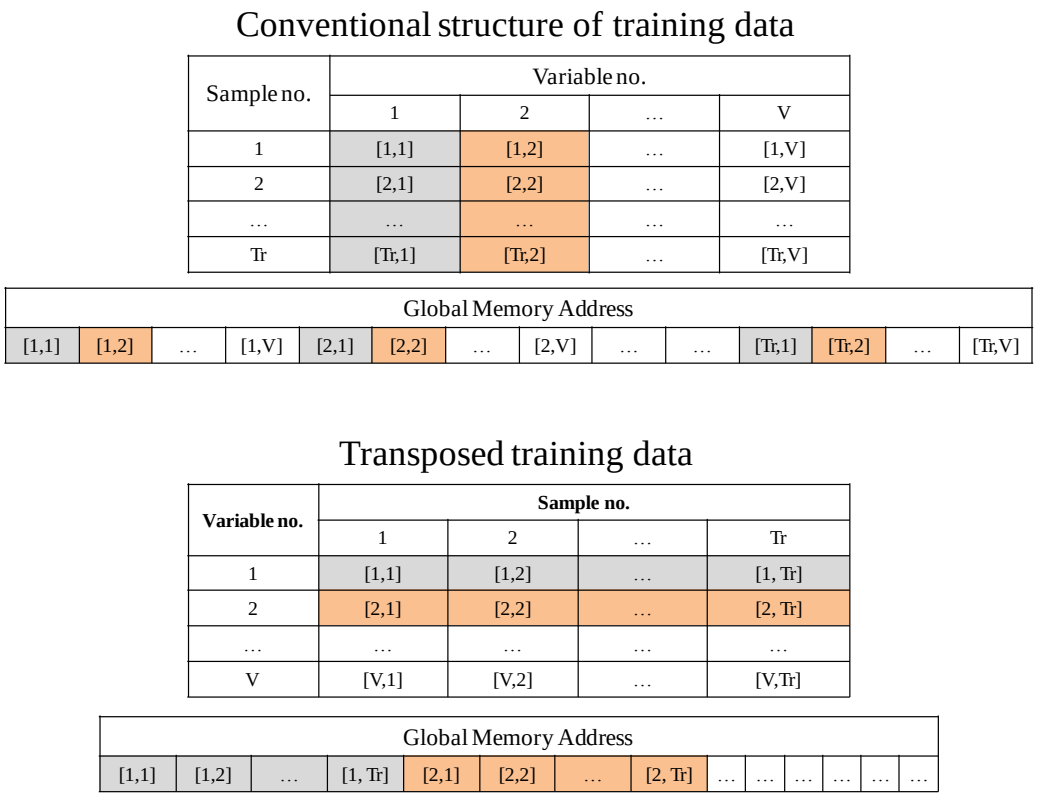

Figure 4-4: Accessing patterns of consecutive threads (same color) before and after data transposition for training dataset.

Conventional structure of eigenvectors

Variable no.	Eigenvalue no.			
	1	2	…	*V*
1	[1,1]	[1,2]	…	[1,*V*]
2	[2,1]	[2,2]	…	[2,*V*]
…	…	…	…	…
V	[*V*,1]	[*V*,2]	…	[*V*,*V*]

Eigenvectors in shared memory

Bank no.				
	1	2	…	*V*
1	[1,1]	[2,1]	…	[*V*,1]
2				
…				
V				

Figure 4-5: Conventional data structure of eigenvectors and the corresponding bank locations in CUDA shared memory.

Transposed eigenvectors

Eigenvalue no.	Variable no.			
	1	2	…	*V*
1	[1,1]	[1,2]	…	[1,*V*]
2	[2,1]	[2,2]	…	[2,*V*]
…	…	…	…	…
V	[*V*,1]	[*V*,2]	…	[*V*,*V*]

Transposed eigenvectors in shared memory;
no. of variables = no. of banks

Bank no.				
	1	2	…	*V*
1	[1,1]			
2	[2,1]			
…	…			
V	[*V*,1]			

Transposed eigenvectors in shared memory;
no. of variables > no. of banks

Bank no.				
	1	2	…	*V*
1	[1,1]	…		
2	[2,1]	[*V*,1]		
…	…			
V	…			

Figure 4-6: Transposed eigenvectors and the corresponding bank locations in shared memory.

On top of that, a series of numerical experiments are carried out to determine the optimal block size for each CUDA kernel. Note that we use the smallest block size, i.e. 32 threads, for the PCA modeling kernel due to the limited amount of data parallelism. The results shown in Figure 4-7 to Figure 4-10 are obtained using the process data of TE Challenge problem which will be discussed in subsequent section. An overview of the proposed hybrid GA-PCA algorithm, which consists of 5 CUDA kernels along with 4 data copying operations, is given in Figure 4-11.

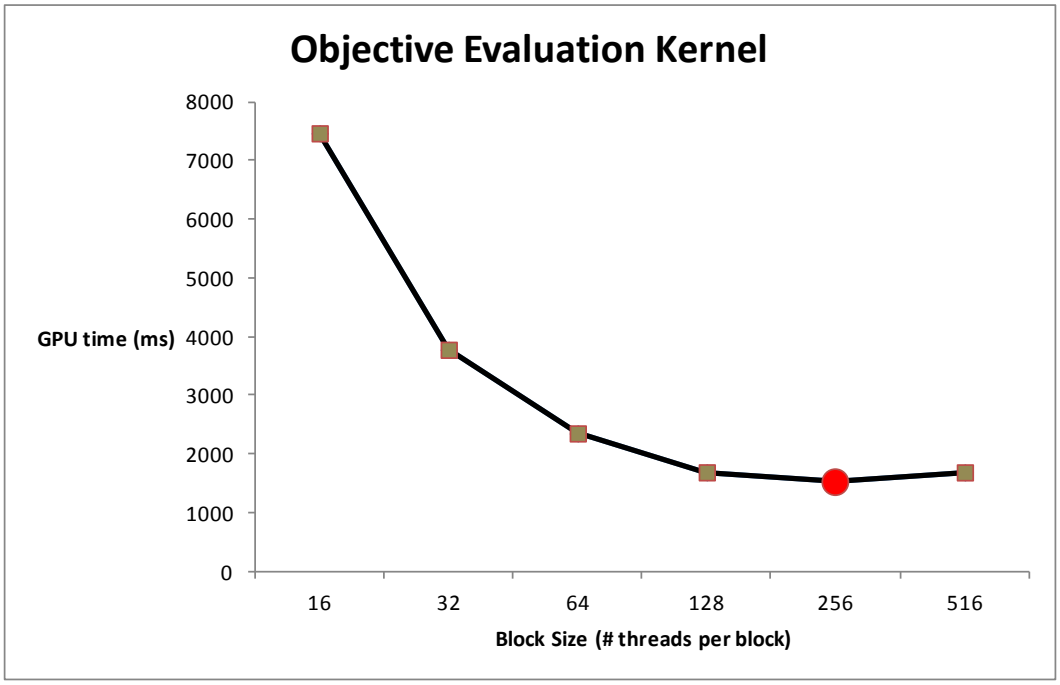

Figure 4-7: CUDA kernel computing times of objective evaluation at various block sizes; optimal block size is 256.

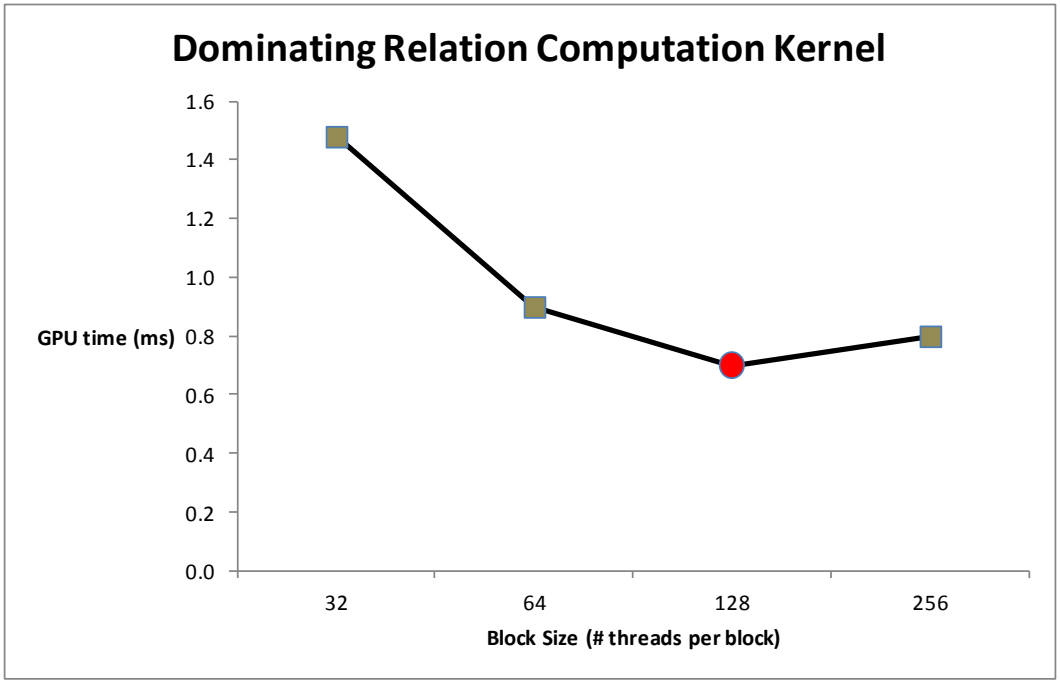

Figure 4-8: CUDA kernel computing times of dominating relation determination at various block sizes; optimal block size is 128.

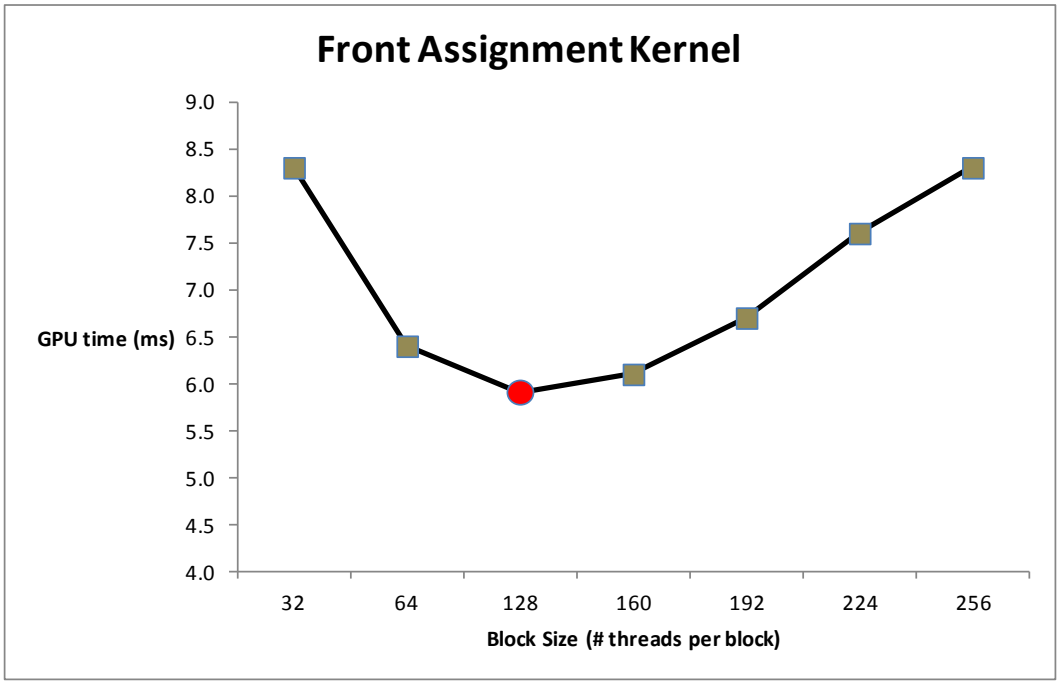

Figure 4-9: CUDA kernel computing times of front assignment at various block sizes; optimal block size is 128.

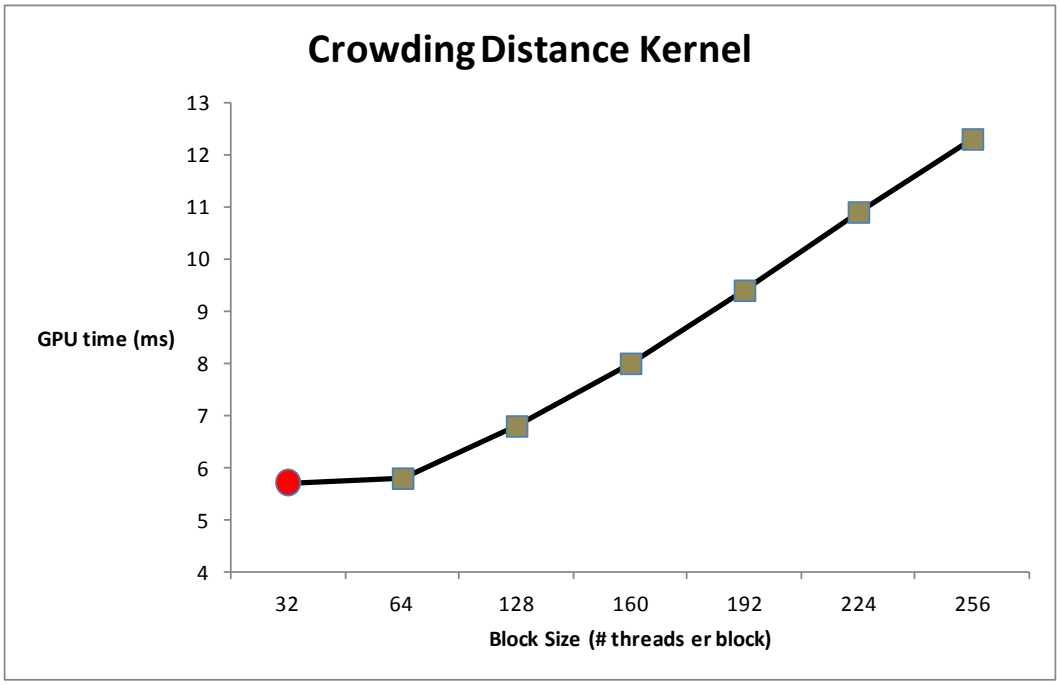

Figure 4-10: CUDA kernel computing times for crowding distance computation at various block sizes; optimal block size is 32.

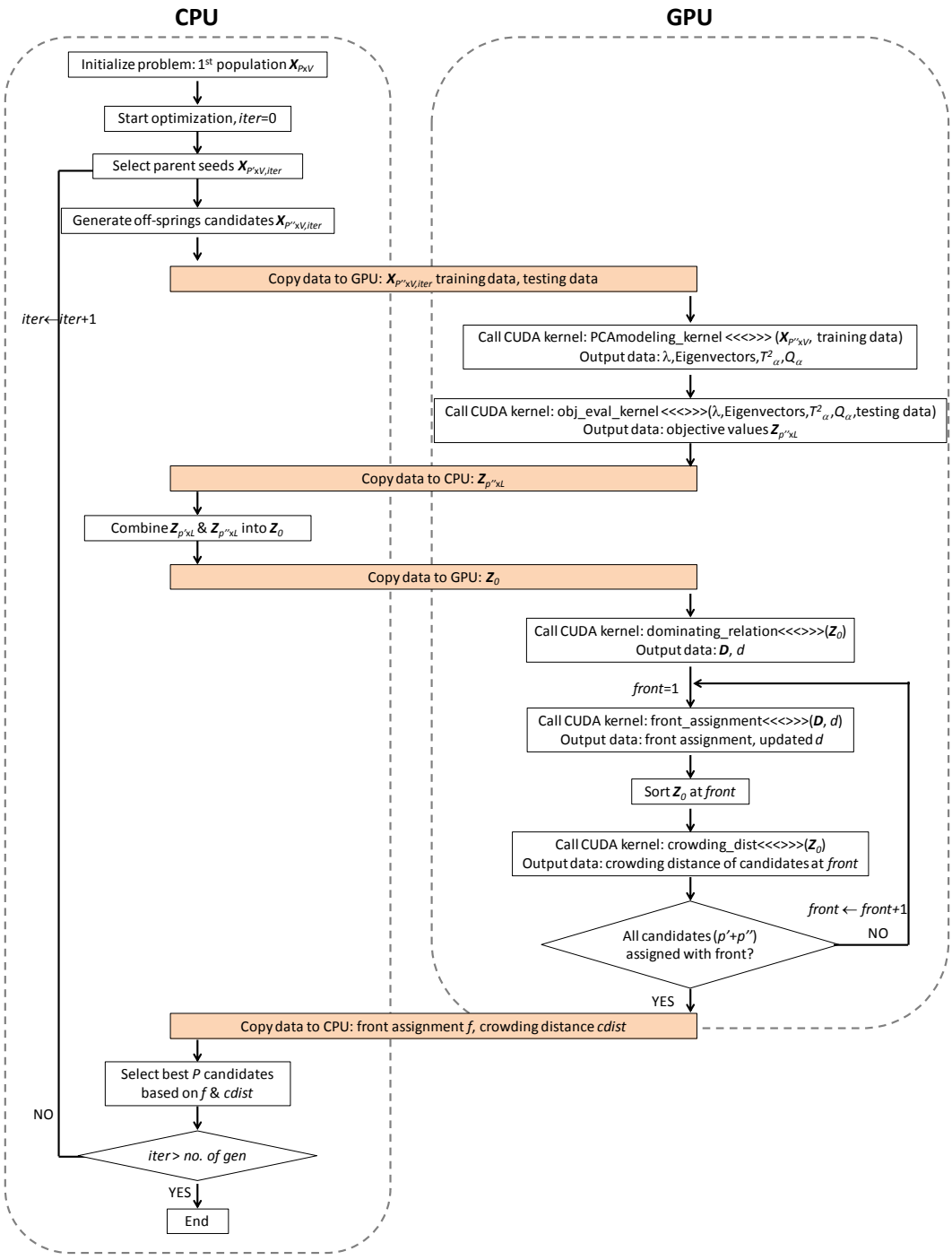

Figure 4-11: The proposed hybrid CPU-GPU-GA-PCA algorithm.

4.4. Case Study: The Tennessee Eastman Challenge Problem

In this section, the proposed hybrid GA-PCA algorithm is tested on the TE process, though an offline variable selection approach is implemented, the achieved

computational speedup demonstrates good promises for real-time application. The TE process is a benchmark simulation proposed by Vogel et al. (1993) to provide a realistic chemical industrial process for studying and evaluating process control technology. The process consists of five main sections including reactor, condenser, separator, compressor and stripper, and it involves two simultaneous gas-liquid exothermic reactions. Four reactants, A, C, D and E, together with the presence of an inert B are required to produce two products, G and H, and a by-product F. The process flowsheet shown in (Vogel et al., 1993) is re-produced in Figure 4-12. It contains 22 process measurements, 19 composition measurements and 11 manipulated variables, thus a total of 52 process variables are involved; in this case study, we have focused on 32 variables. The training and test datasets were obtained from http://brahms.scs.uiuc.edu/, wherein 500 samples of normal operating condition are used as training data, while 960 samples from the normal operation and 800 samples from each of the 21 programmed faults are used as testing data. In this variable selection problem, our goal is to select the best subset of variables from the 32 variables which are most sensitive to the 21 faults. As the objective function is defined based on the normal and faulty classification, identification of the fault type is not required.

The computing system used in the experiments is 64-bit Intel Xeon CPU with processor speed of 3.20GHz and RAM of 12.00GB memory size; the GPU co-processor is an Nvidia Quadro 2000 card. In the following, we analyze the results obtained from the proposed CPU-GPU-GA-PCA algorithm, and compare it with that achieved by the original sequential approach.

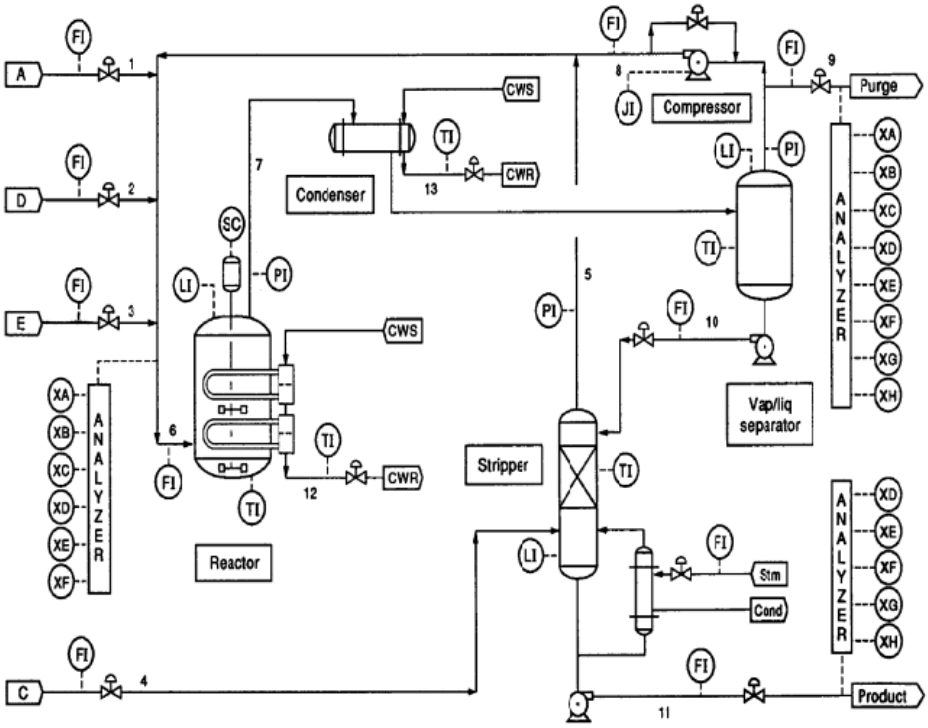

Figure 4-12: Tennessee Eastman process flow sheet[8]

4.4.1. **Comparison of monitoring performance**

It is worthless to have an extremely efficient algorithm that has poor quality, thus we first need to ensure the monitoring quality of PCA model is not compromised with the use of GPU. At the end of the GA-PCA optimization, there will be *P* surviving solutions output from the algorithm. To guarantee for comparable monitoring quality, the *P* final solutions obtained from the sequential CPU approach and the hybrid approach should have comparable objective values, i.e. the number of variables chosen and the cumulative error rates. As it can be seen from Figure 4-13, the hybrid approach provides equivalently good monitoring performance as compared to the CPU method; this result is obtained at the end of 100 GA generations using 100 populations. The

[8] Courtesy: (Downs & Vogel, 1993)

monitoring quality of the hybrid approach has also been verified at larger GA generations and population size, as shown in Figure 4-14; using 1000 generations and 1000 populations.

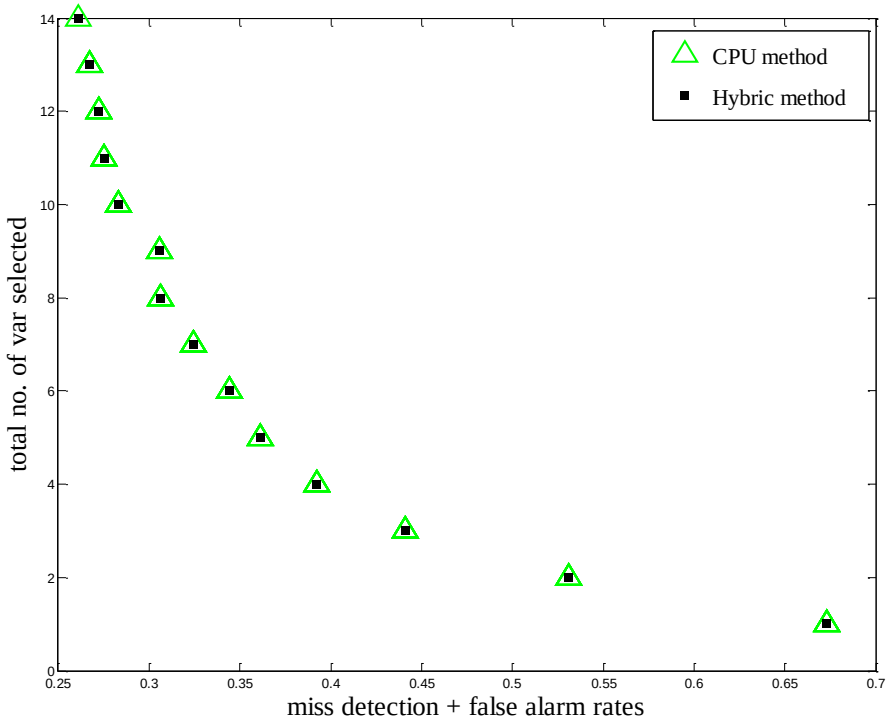

Figure 4-13: Objective values obtained by the CPU and hybrid methods, at the end of 100 GA generations, with 100 populations.

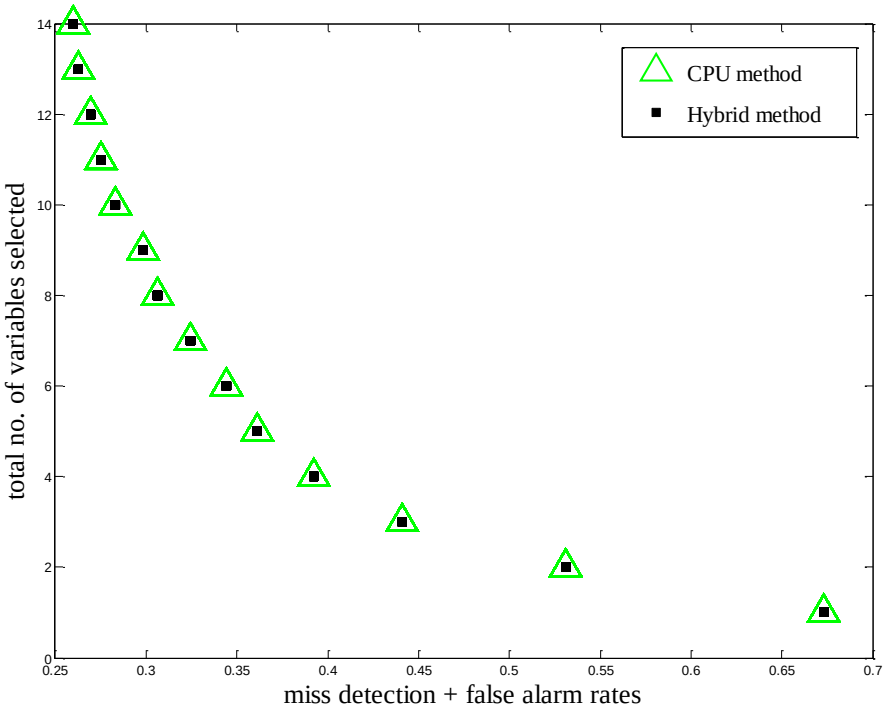

Figure 4-14: Objective values obtained by the CPU and hybrid methods, at the end of 1000 GA generations, with 1000 populations.

4.4.2. Efficiency improvement achieved by hybrid CPU-GPU approach

We now evaluate the efficiency performance of the hybrid CPU-GPU-GA-PCA variable selection algorithm, which is constructed based on the proposed systematic procedure. Note that all results reported in following discussions are the average values of three duplicated experiments. As illustrated in Figure 4-3, the computing time consumed by different portions of the algorithm vary as the parameters change. Therefore, we analyze and discuss the effect of two key parameters, which are the number of GA generations and population size, on the achieved efficiency enhancement.

4.4.2.1. **Effect of number of generations**

Several observations can be made from the Figure 4-15 which shows the total computation time consumed by the CPU method and the hybrid method, at the end of various generations using 1000 populations. Literally straight lines are seen for both methods, indicating that the amount of time required in each generation is consistent throughout the optimization course, given the same population size. This can be explained by the fact that the computational workload required by same amount of candidate solutions in each optimization cycle would be similar. Firstly, determination of the relationship matrix $\boldsymbol{D}_{pxp}$ involves computational complexity $O(P^2 \times L)$, which is a function of population size. Also, same training and testing datasets are used throughout all GA generations, so computational workload involved in the PCA modeling or objective evaluation would not be different significantly.

By parallelizing the 3 computationally expensive tasks (i.e. PCA modeling, objective evaluation and non-dominated sorting) using GPU, it is expected that the hybrid method could complete the Sim-Opt problem in a shorter time than the sequential counterpart. The reduction of computing time can be clearly seen from Figure 4-15, where the red line representing hybrid method is significantly more gradual and flat as compared to the blue line. The speedup ratios computed according to Equation (3-2) are plotted in Figure 4-16. Since the computing time of both CPU and hybrid methods scale linearly as the number of generations increases, the speedup values are approximately constant. From Figure 4-16, it can be observed that about 11 speedups are achieved by the hybrid method at GA generations in the range of [40 1600]. Due to the linearly scaling characteristics of both methods, we expect identical efficiency performance to be achieved by the hybrid method, at larger generations.

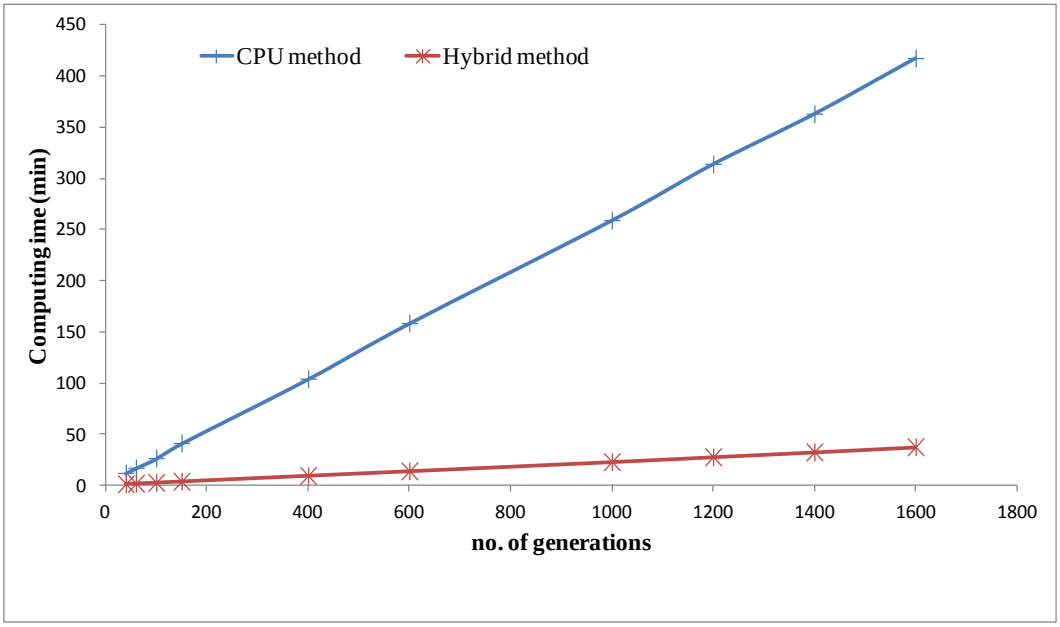

Figure 4-15: Computing time of the CPU and hybrid methods measured at various numbers of generations, with 1000 populations.

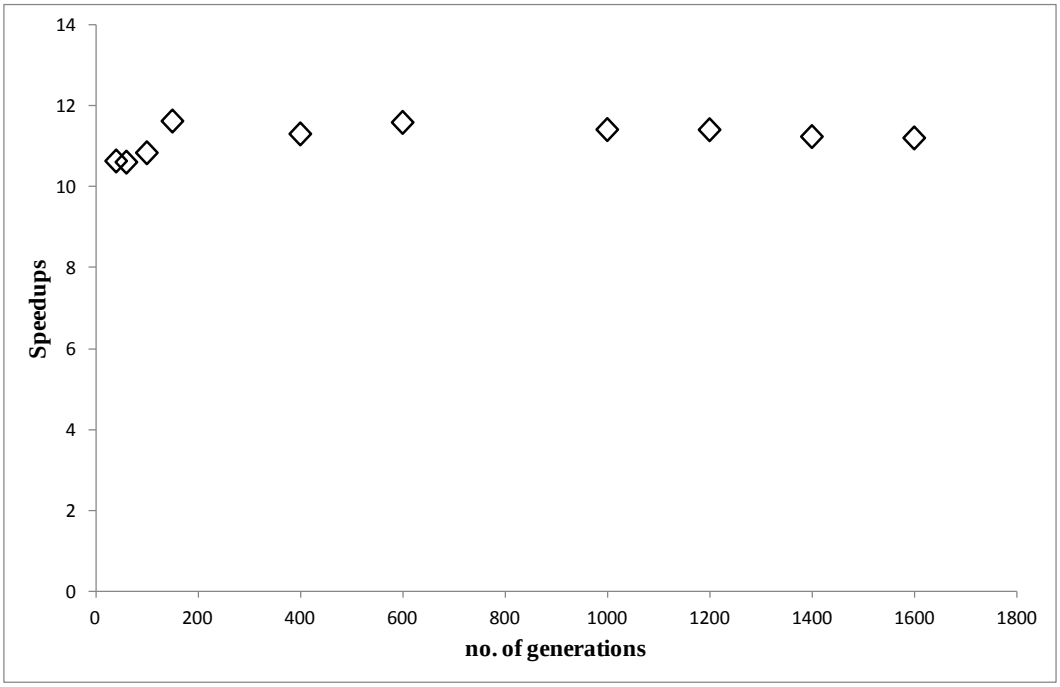

Figure 4-16: Speedups achieved by the hybrid method at various numbers of generations, with 1000 populations.

4.4.2.2. **Effect of population size**

Here, we investigate the effect of population size on individual tasks, as well as on the overall performance. In the original sequential GA-PCA algorithm, the

computation of $\boldsymbol{D}_{pxp}$ in NSGA-II involves 2 nested loops with *P* elements in each loop, so as *P* increases the computing time is expected to increase in a nonlinear manner. On the other hand, as the computing time required by the covariance matrix computation in PCA modeling is dependent only on the number of variables selected; and time consumed in score value computation and sample classification in objective evaluation depend on the testing data and number of variables selected. These two tasks have linear relation between *P* and computing time. Overall effect of *P* on total computing time is therefore determined by the strengths of the linear and nonlinear effects. As shown in Figure 4-3, NSGA-II dominates at large population size, while PCA modeling and objective evaluation consumes a large portion of total time at small populations. Such remark is consistent with the observation obtained from Figure 4-17, where the computing time of CPU method (represented as blue crosses) resembles a straight line at small *P* and gradually shows off the nonlinearity characteristic at larger *P*.

With GPU parallelization, the linear and nonlinear characteristics would be diluted, where the dilution factor depends on CUDA kernel design. In both CUDA kernels of PCA modeling and objective evaluation, *P* candidate solutions are mapped to *P* CUDA blocks. Supposing *P* is a large number and GPU resource has limitation, there will only be C_p number of blocks executing concurrently, at any one time. As *P* increases by a factor of *f*, the computing time required by the CUDA kernels will multiply by $\frac{f}{C_p}$ instead of f; the latter is only true for sequential method. Given the same GPU device, the dilution factor C_p is constant, and the effect of population size on computing time is still linear for the two kernels. Next, we look at the nonlinearity contributor which is NSGA-II. In this CUDA kernel, the key computation $\boldsymbol{D}_{pxp}$ is achieved by mapping *P* rows to *P* blocks and *P* columns to *T* threads. Hence, as *P* increases by a factor of *f*, the increase of

computing time for $\boldsymbol{D}_{pxp}$ computation is approximately $\frac{\left[\frac{(P+f)\times(P+f)}{C_p\times T}\right]}{\frac{P\times P}{C_p\times T}}$ instead of $\frac{(P+f)\times(P+f)}{P\times P}$ in sequential algorithm. Given the two-level parallelization i.e. dilution, not only the nonlinearity behavior becomes less apparent, the total computing time will also be reduced significantly. It can be seen from Figure 4-18, by implementing the sorting in GPU, the nonlinearity behavior is no longer noticeable at the tested range of population size.

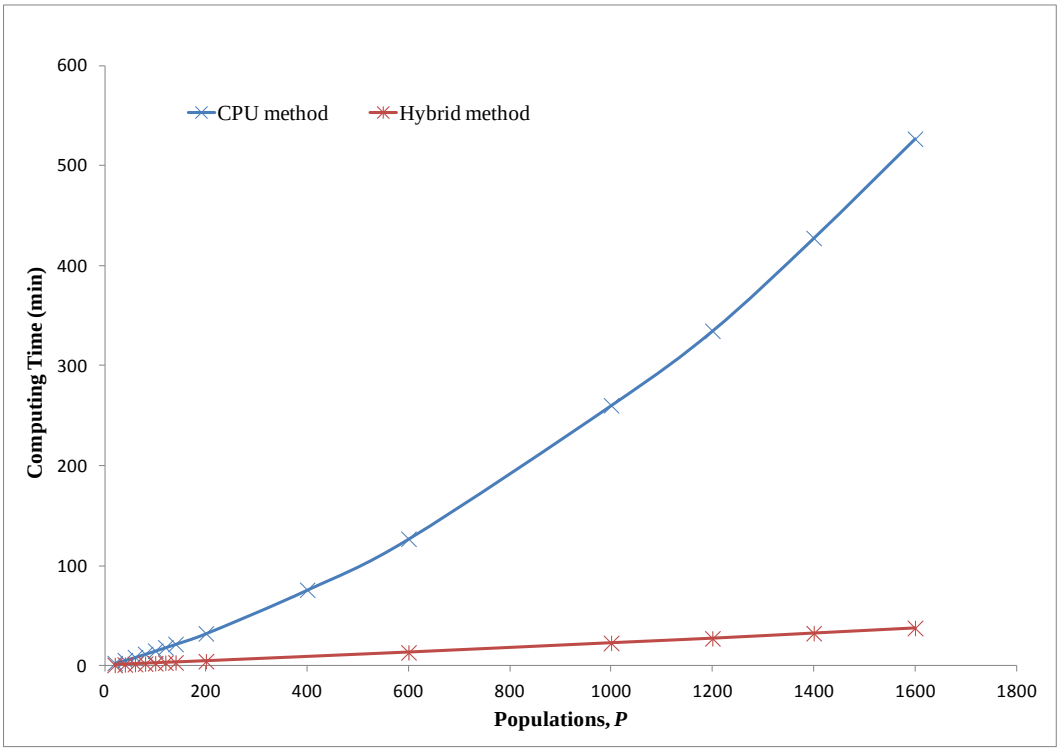

Figure 4-17: Computing time of the CPU and hybrid methods measured at various population sizes, with 1000 GA generations.

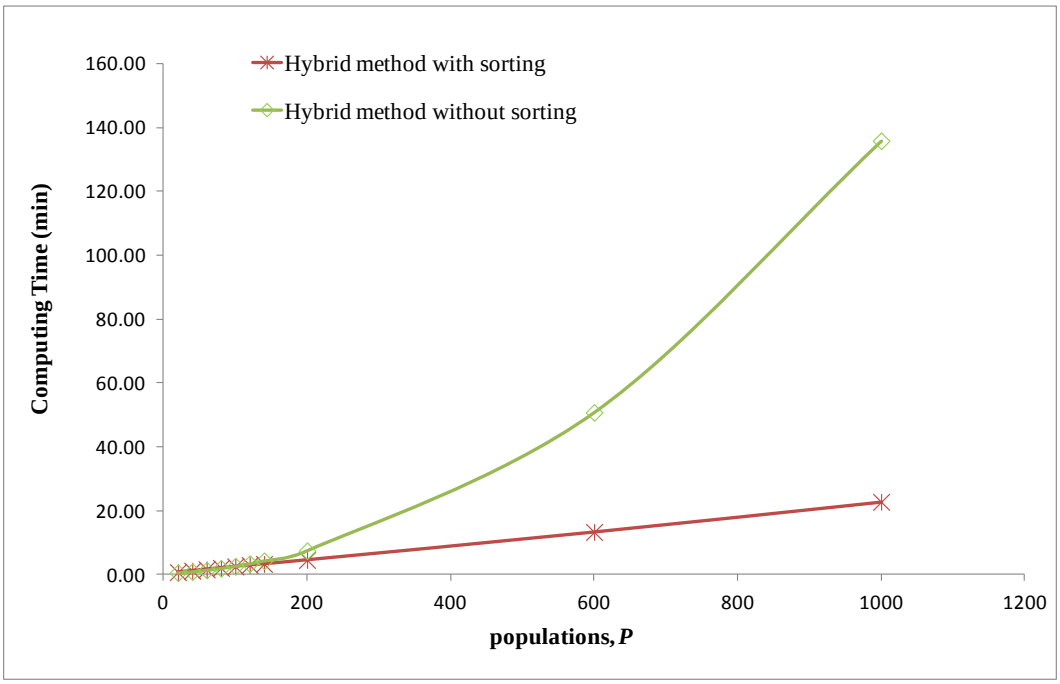

Figure 4-18: Comparison of computing time of two hybrid methods, one with and the other one without GPU-based sorting; measurements are done at various population sizes, with 1000 GA generations.

Finally, we examine the efficiency performance attained by the proposed hybrid method. Based on the computing time of CPU and hybrid methods measured at end of 1000 generations (as given in Figure 4-17), the corresponding speedup ratios are plotted on Figure 4-19. The quickly increasing speedup ratios at population size below 200 can be attributed to one possible cause. That is GPU resources are under-utilized when *P* is small, so when it is supplied with more *P*, it is able to complete the job using approximately same amount of time along with additional data transferring time. Due to the long computational time required by CPU method, for example it takes more than 8 hours (or 24 hours for 3 duplicates) to complete an experiment using 1600 populations and 1000 generations, thus we evaluate the efficiency performance for larger populations at a smaller number of generations i.e. 100. An approximately linear speedup relation can be observed from Figure 4-20; the computing time for both the CPU and hybrid methods is also shown. At 7500 populations, the proposed hybrid

method greatly reduces the total computing time from 43 hours to 26 minutes, and this is equivalent to 102 speedups. Hence, the proposed hybrid GA-PCA algorithm offers great advantages in control and monitoring applications which require continuous model optimization, due its capability of capturing timely and accurate process behavior change.

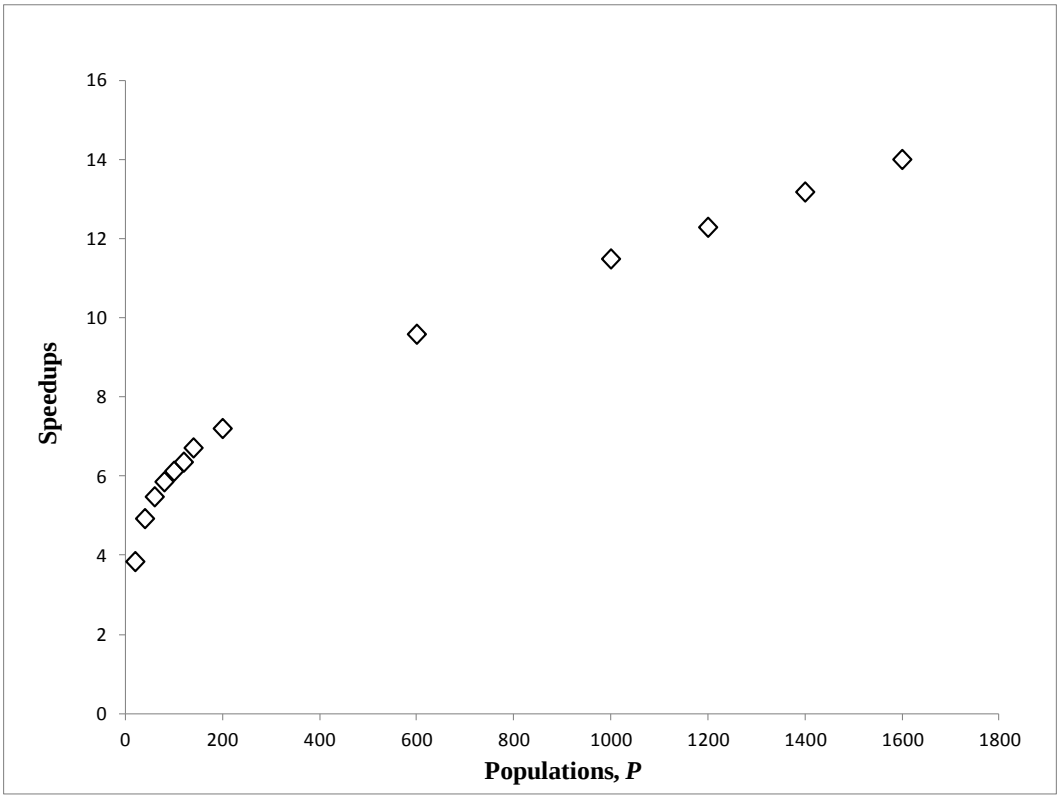

Figure 4-19: Speedups achieved by the hybrid method at various population sizes, with 1000 GA generations.

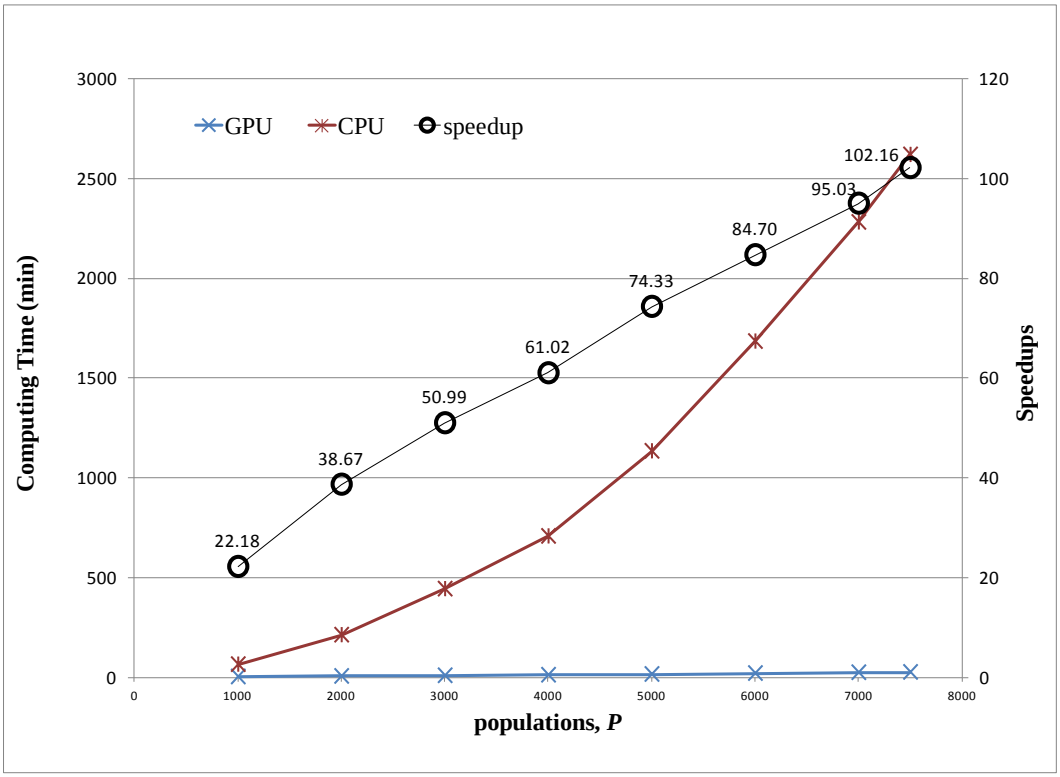

Figure 4-20: Computing time for the CPU and hybrid methods, along with the corresponding speedups (represented in black circles), at various population sizes and 1000 GA generations.

4.5. Summary

Sim-Opt is a special class of optimization techniques which are capable of dealing with complex process system as well as uncertainty. Without sacrificing process information unnecessarily, applications of Sim-Opt are strictly computationally limited due to the need of processing large amount of data or dealing with complex simulation model. In order to address the practicality issue of Sim-Opt, we propose a systematic procedure for developing efficient hybrid Sim-Opt algorithm using GPU parallelization. A step-by-step discussion of the systematic procedure is illustrated on a variable selection algorithm, which is originally proposed by Ghosh et al. (2014) for solving a process monitoring problem. The variable selection algorithm is regarded as a Sim-Opt technique by treating GA as the optimization tool and PCA as the simulation model. Based on the proposed approach, 5 CUDA kernels are constructed for the 3 tasks,

namely PCA modeling, objective evaluation, and non-dominated sorting. The primary data parallelisms exploited in these kernels include mapping CUDA blocks to candidate solutions, and mapping CUDA threads to training samples, testing samples or candidate solutions. The performance of the developed hybrid CPU-GPU-GA-PCA algorithm is tested as an offline variable selection method on the TE process with 32 process variables. The GPU-based method shows comparable monitoring performance, in terms of miss detection and fault alarm rates, to the original sequential method. Furthermore, the computational efficiency of the GA-PCA algorithm is greatly improved in the hybrid approach, with speedups as high as 102. The significant improvement in computational speed promises for the practical use of Sim-Opt algorithm in solving real-time or continuous optimization problems.

Chapter 5. Hybrid CPU-GPU Multivariate Image Analysis Algorithm

Image analysis is an important sub-topic of pattern recognition which involves the extraction of useful information from digital images by means of image processing techniques. In the context of chemical processes, the extracted information is mainly used for predicting, monitoring, or controlling purpose. Recently emerged image analysis techniques called multivariate image analysis (MIA) are capable of handling multiple univariate images at once, thus uncovering more informative and useful pattern and knowledge. In contrast to traditional image analysis methods, MIA algorithms deal with much larger input data size and require the use of multivariate statistical tools such as PCA and PLS. While the growing advances of imaging technology offer high resolution (i.e. large amount of pixels per frame) and high frame rate images at low cost, the computational demand for processing those images is huge. Hence, the task of extracting information in real-time for effective process control and monitoring is extremely difficult, due to the computational complexity involved in MIA and also the large data size of input images.

Although there have been a number of GPU-accelerated MIA approaches proposed in the literature, none of them is meant for chemical process applications. As the input images (in terms of color scale), image enhancement expectation, and the types of pattern or object to be determined from the target images are typically application-dependent. The existing GPU-based methods might not be directly useful in solving chemical process problems. In this respect, our main contribution is to develop a

Related publications:

Lau Mai Chan, Rajagopalan Srinivasan. A Hybrid CPU-Graphics Processing Unit (GPU) Approach for Computationally Efficient Simulation-Optimization. Manuscript submitted for publication.

Real-time Particle Size Estimation for Crystallization Processes through GPU-based Multivariate Image Analysis, AIChE Annual Meeting 2013, San Francisco, CA.

GPU-based MIA algorithm which is applicable to chemical process control and monitoring.

Specifically, we aim to address the issue of high computational cost of MIA algorithm by developing a hybrid CPU-GPU-MIA algorithm using GPU parallelization technology. A major motivation is that the massive amount of data parallelism offered by GPGPU fits perfectly to the large-scale image processing algorithm such as MIA, whereby huge amount of pixels can be processed simultaneously. The organization of the remainder of this chapter is as follows. In Section 5.1, we describe a MIA method which is proposed for in-situ particle size estimation during crystallization process. Subsequently, we analyze individual steps of the MIA method in Section 5.2. Based on their contributing computational cost and amount of data parallelism, several GPU tasks are identified. Some important details including CUDA kernel design, required data transfer between CPU and GPU memory, and optimization opportunity are also provided. In Section 5.3, the proposed hybrid MIA algorithm is implemented and evaluated on a case study, which involves a batch crystallization of monosodium glutamate monohydrate (MSG) for a period of 23 hours.

5.1. MIA for In-Situ Particle Size Estimation of Crystallization Process

The capability of estimating crystal size distribution in real-time is important for effective control and optimization of particulate processes. Appropriately controlled crystal size distribution not only ensures high efficiency of downstream operations like filtering, drying and formulation, it also safeguards the efficacy of final product which is in crystal form. In order to achieve fast online measurement of crystal size, automated image analysis has recently been developed (Sarkar et al., 2009). The method has shown

satisfactorily accurate estimation of particle size distribution. However, this comes at the cost of computational efficiency. The minimum requirement for real-time particle size estimation is that computational speed of the MIA algorithm is at least on par with the image generation speed.

The MIA method proposed by Sarkar et al. (2009) consists of several tasks, which include feature extraction, construction of statistical image model, image segmentation, post-segmentation image analysis, and boundary refinement. Each step is detailed as below.

5.1.1. **Feature extraction**

In the context of image analysis, various features which are useful characteristics can be extracted from raw input image. In this work, we focus on three features which are range, standard deviation and entropy; mathematical expressions given in (Sarkar et al., 2009) are reproduced below,

$$f_{range}(x,y) = \underset{\substack{i\in(x-B,x+B)\\ j\in(y-B,y+B)}}{max}\{f(i,j)\} - \underset{\substack{i\in(x-B,x+B)\\ j\in(y-B,y+B)}}{min}\{f(i,j)\} \quad (5\text{-}1)$$

$$f_{stdev}(x,y) = \underset{\substack{i\in(x-B,x+B)\\ j\in(y-B,y+B)}}{stdev}\{f(i,j)\} \quad (5\text{-}2)$$

$$f_{entropy}(x,y) = -\sum_{\substack{i\in(x-B,x+B)\\ j\in(y-B,y+B)}} p(f(i,j))\, log_2\, p(f(i,j)) \quad (5\text{-}3)$$

where, *B* defines the size of filter mask (in this work, *B* = 1 which is equivalent to a 3 by 3 square mask); *i* and *j* represent the vertical and horizontal pixel location, respectively; (*x*, *y*) denotes the pivot and also the output pixel location for feature calculation; $p(f(i,j))$ is the histogram of intensity levels in the mask. Given a grayscale image with resolution *X* by *Y*, there will be 3 individual feature images generated from the raw univariate image.

In order to capture the spatial relationships among pixels, a shifting operation is performed on each of the feature image. This is achieved by moving the pixels of feature image r_s space (i.e. shifting radius) away in the specified direction. Here, we use r_s=1 and d=8, so there will be 8 shifted images produced for every feature image; the 9 images are collectively referred as a multi-way image. The 8 shifting directions employed in this work are depicted in Figure 5-1. The resulting 27 images (i.e.$(8 + 1) \times 3$) are regarded as a multivariate image containing 27 variables. The application of a suitable multivariate statistical is discussed next.

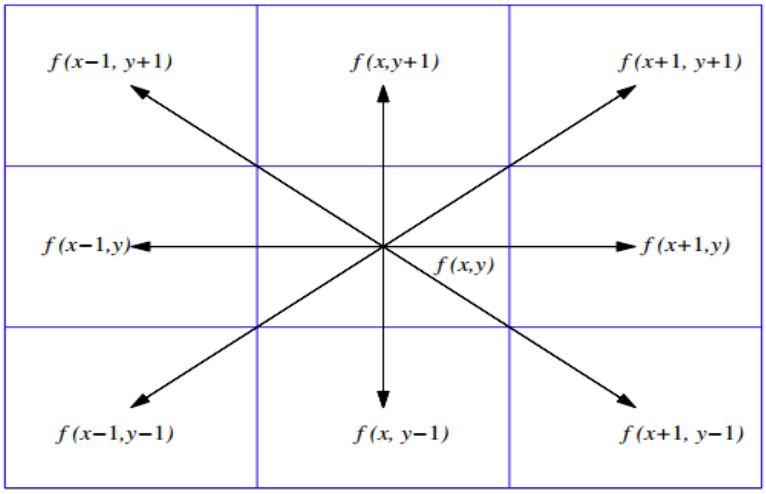

Figure 5-1: The 8 shifting directions used in MIA algorithm[9].

5.1.2. Multi-way PCA and statistical image model

For effective image segmentation in later stage, statistical models need to be constructed as they enable straightforward thresholding. Multi-way PCA can be used to transform a multivariate image into an image model, which serves as a concise and noise-free representation of the original input image. Multi-way PCA works as the same

[9] Courtesy: (Sarkar et al., 2009)

manner as traditional PCA method, except that the multivariate input data is required to be first unfolded into a 2-dimensional matrix. Unfolding is achieved by opening up the $X \times Y$ pixels of each image into single dimension. As a result, a 2D matrix with $(X * Y)$ rows and 27 columns is obtained. Subsequently, N_p principal components will be chosen during PCA analysis which results in a score matrix of size $(X * Y) \times N_p$. As we only retain the first principal component (i.e. N_p = 1), the score matrix can be easily converted back to original image size of $(X * Y)$ which is called pseudo-image. In cases where N_p is more than one, Hotelling's T^2 statistics as given in Equation (4-3) can be used to reduce score matrix from size $(X * Y) \times N_p$ to $(X * Y) \times 1$.

5.1.3. **Image segmentation**

As has been discussed early, the main goal of image segmentation is to locate objects using thresholding strategy, and the outcome is a binary image. First of all, a global threshold value ψ_s is obtained offline from a background image. The value is computed by multiplying a user-specified parameter α to the standard deviation determined from the associated pseudo-image. During the online phase, the pseudo-images obtained from in-situ images are segmented into objects and background regions, based on the threshold value.

5.1.4. **Post-segmentation image analysis**

At this stage, a flood-fill operation is first used to fill up the missing holes , which is then followed by a morphological opening operation to remove all objects which are below a specified pixel area ψ_a. After removing objects touching the image boundary, remaining objects are recognized as particles by assigning with unique object labels. Lastly, the exterior boundary of the shortlisted particles is identified.

5.1.5. Boundary refinement and particle size estimation

Characterization of particles from image analysis which provides important information such as particle size and maximum chord length (MCL) requires clean and refined boundary. It begins with converting the boundary pixels identified from post-segmentation to a series of Fourier descriptors of *K* frequencies, using discrete Fourier transform technique. Note that lower frequency components of Fourier descriptors describe the macro shape of object while higher frequency components provide fine details of the boundary. As the macro shape of object plays an important role in characterizing particle, a specified number N_{FC} of low frequent components are used ($N_{FC} < K$) for reconstructing the boundary on the original image 2D plane, using inverse Fourier transform. The refined boundary can then be used to determine object area and MCL. In this work, object area which is measured in terms of number of pixels serves as the comparison basis for evaluating the quality of our proposed hybrid approach. Hence, the detailed discussion of MCL which involves the concept of signature curve and antipodal angle threshold is omitted here; interested readers are referred to the original article (Sarkar et al., 2009).

5.2. GPU Implementation of MIA

In this section, we develop a GPU-based MIA algorithm following the 5-step approach as presented in Section 4.2. We discuss the individual steps in the context of MIA algorithm as follows.

Computing time examination: It can be seen from the pie chart presented in Figure 5-2 that the most computationally demanding tasks are feature extraction, PCA modeling and segmentation, which account for 91% of total computing time. Hence, these 3 tasks will be examined for GPU parallelizability in the following step.

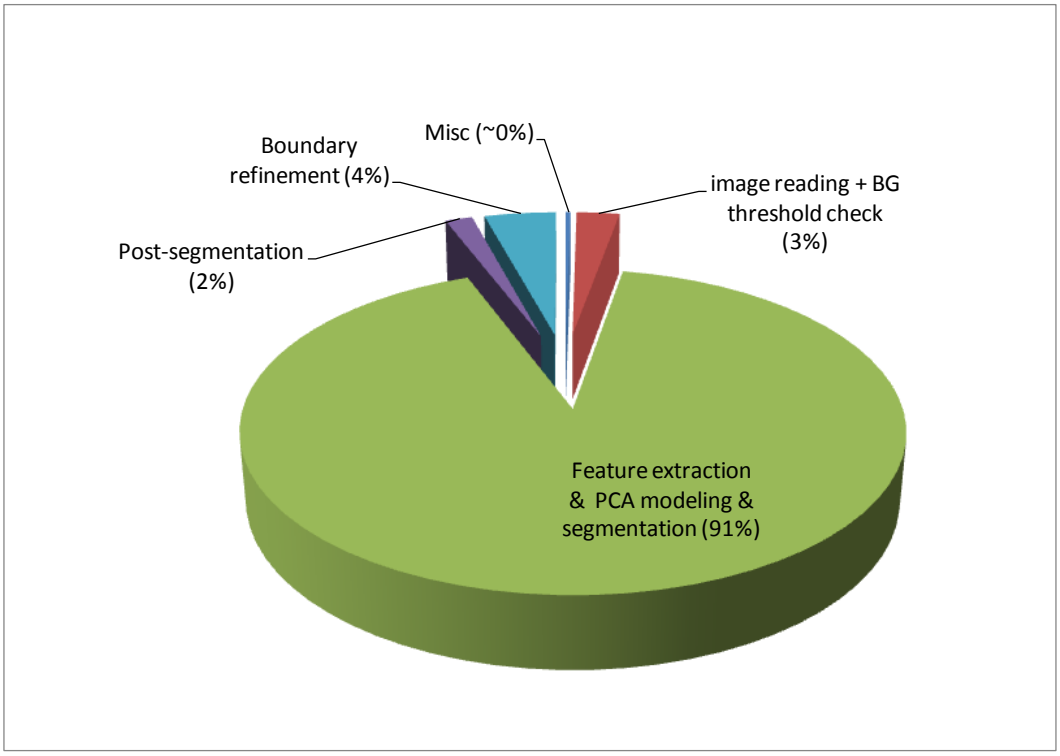

Figure 5-2: Pie chart showing the computing time distribution of the original MIA algorithm; computing time measurements are based on averaging of 50 images.

GPU task selection: In this step, the shortlisted tasks are examined for amount of data parallelism and GPU-compatibility. Since feature extraction and segmentation involve mainly independent pixel operations, there is enormous amount of data parallelism. Furthermore, they require only simple computations such as comparison, summation and division which are perfectly supported by GPU, as can be seen in Equations (5-1), (5-2) and (5-3).

Another costly task under consideration is PCA modeling based on a matrix of size $(X * Y) \times 27$. It is commonly advisable to normalize the input matrix prior to the building of PCA model for avoiding biases towards variables with large magnitude. Both the normalization of feature images and the computation of covariance matrix offer significant amount of data parallelism, wherein massive data parallelism is available on

independent column or element operations, respectively. Although the determination of eigenvalues and eigenvectors is not inherently rich in data parallelism, there are still GPU algorithms reported to have better performance than the CPU counterpart. In this work, we employ the GPU-NIPALS-PCA algorithm proposed by Andrecut (2009). More details will be given during the discussion of CUDA kernel design.

Data transfer consideration: Since the 3 selected GPU tasks are back-to-back processes using output data from preceding task as their input, as illustrated in Figure 5-3. The data required to be transferred from CPU to GPU mainly include the raw input image and parameters such as shifting directions, mask size, threshold, range threshold and number of principal components. The grayscale input image is stored in GPU global memory instead of texture memory, because there is no spatial-localized accessing pattern, as will be elaborated in later paragraphs. Whereas, the parameter values can either be stored in constant memory or global memory.

As it can be seen from Figure 5-3, intermediate storage spaces in global memory are needed for the 3 feature images, 27 shifted images, covariance matrix and pseudo-image, so as to convey data information across different CUDA kernels. Lastly, the binary image output from segmentation which is stored in global memory will be transferred back to CPU for post-segmentation analysis.

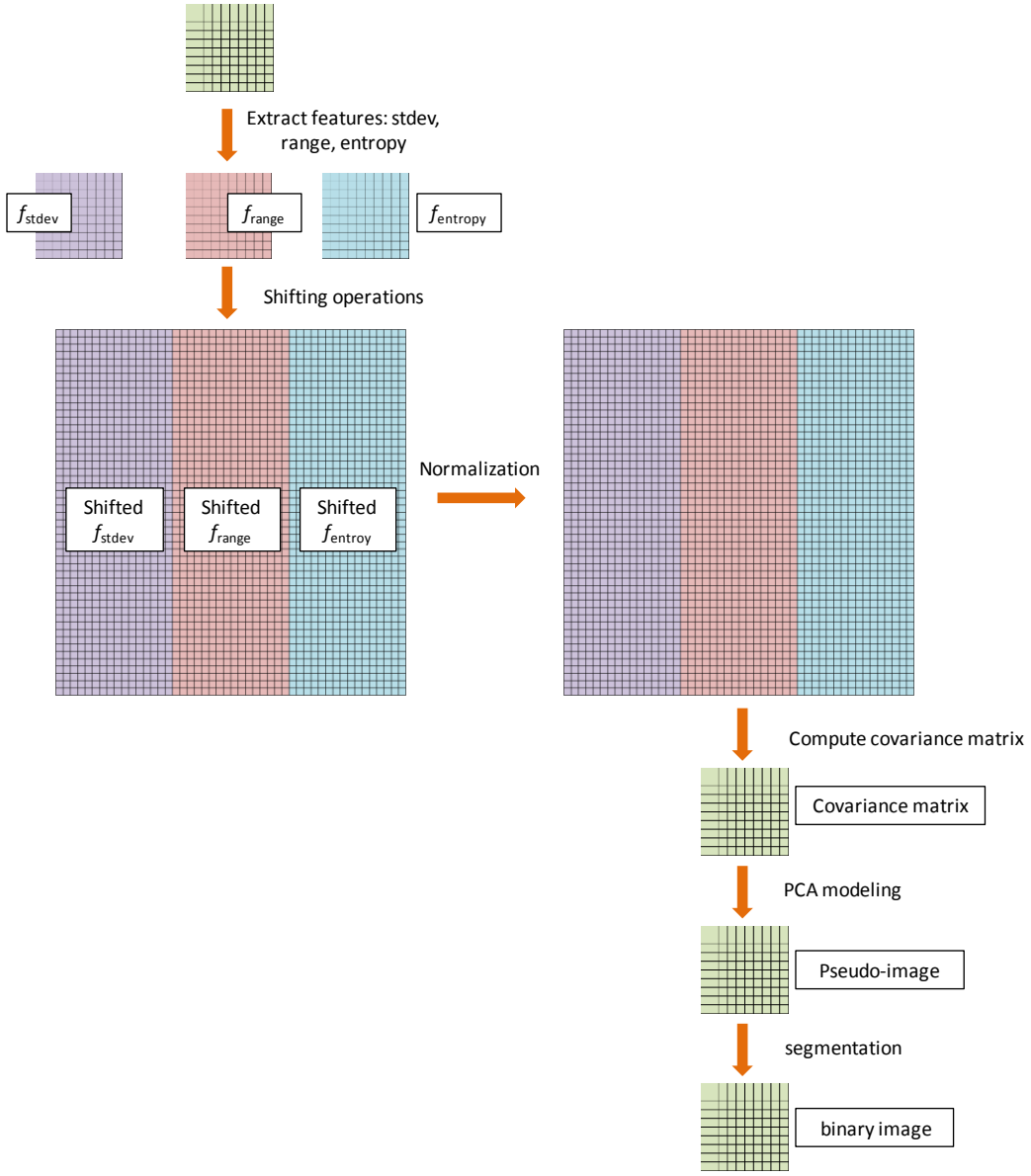

Figure 5-3: Information flow among the 3 expensive tasks of MIA.

CUDA kernel design: First of all, it should be noted that the 3 tasks, i.e. feature extraction, PCA modeling, and segmentation, are strictly sequential in nature as displayed in Figure 5-3. In other words, a minimum of 3 CUDA kernels are required for performing these tasks, each of which serves as an explicit synchronization point. Any of the two consecutive tasks can be combined in the same CUDA kernel only if they share the exactly same thread configuration, and if within-block synchronization is able to

guarantee for results consistency. A detailed discussion is given below, covering various design decisions to be made during GPU programming. These mainly include the optimum number of CUDA kernels for each task, the mapping of CUDA threads and blocks to computational operations, and the types of memory used.

Figure 5-4 shows a standard method of task partitioning for GPU tasks which involve pixel operations like feature extraction and segmentation, whereby arbitrary number of 2D CUDA blocks are mapped to the pixels. In this case, actual thread configuration is mainly limited by GPU resources, particularly the maximum number of threads per block and the shared memory usage. Nonetheless, the total number of redundant threads which are not assigned to any pixels should be minimized. This strategy is slightly different from the two-level partitioning approach mentioned in Chapter 3. The latter is more suitable for situations where there is apparent existence of sub-tasks.

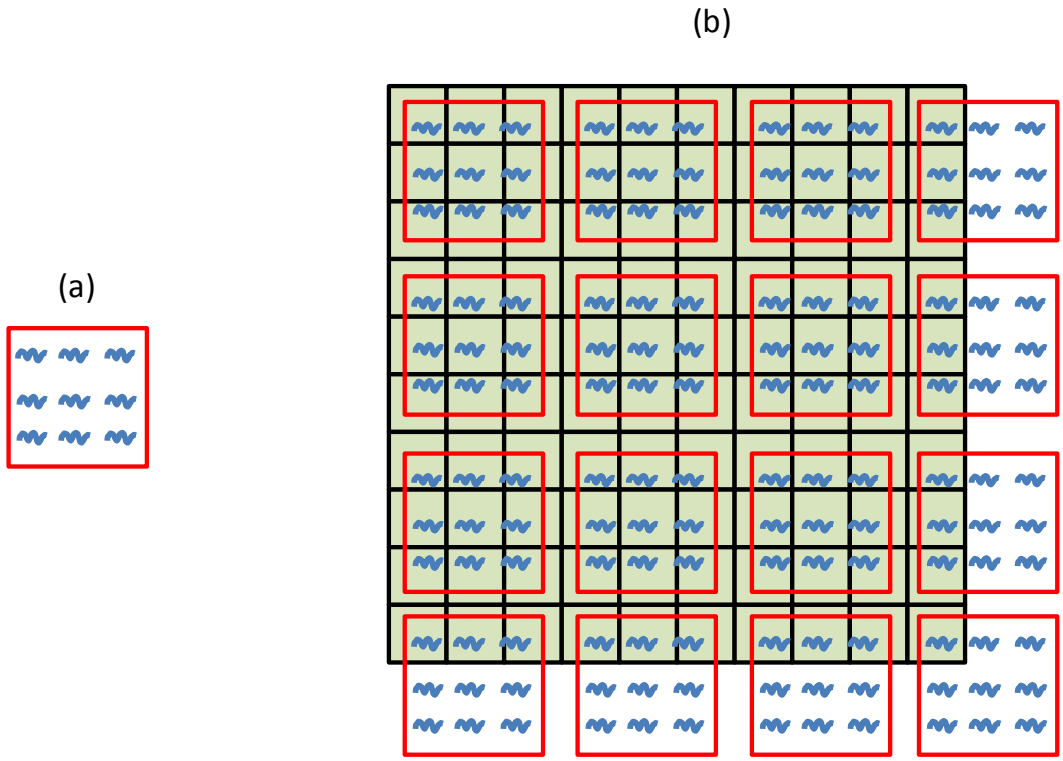

Figure 5-4: Task partitioning scheme for feature extraction and segmentation; (a) representation of a CUDA block containing several threads; (b) mapping of CUDA blocks/ threads to pixel operations.

We now look at the computation performed by individual thread in the feature extraction CUDA kernel. For each feature extraction, i.e. range, standard deviation, entropy, a thread is required to walk through all elements in the 3 by 3 mask, and collect associated information like the minimum and maximum values, summed values, summation of squared values, and intensity counts. Considering the same input image and thread configuration used in all 3 feature extraction process, they can be combined into single CUDA kernel. This provides the advantages of saving memory access bandwidth and reducing kernel call overhead. Local register or shared memory can be used for storing this information, but to avoid register spillage we use shared memory for counting the intensity. For intensity counting, a vector with length equal to 9 is allocated for each thread, in shared memory. Due to the self-synchronizing feature of thread warp, the 32 threads in a warp tend to access the same location in the mask but

consecutive pixel locations in the input image residing in global memory. The resulting kernel is termed as *feature_kernel* for brevity.

In order to ensure results consistency, a separate CUDA kernel (termed as *shifting_kernel*) is used to perform the image shifting process. As presented in Figure 5-5, a particular thread in *shifting_kernel* copies the values from several different pixel locations which are computed by different threads in *feature_kernel*, thus synchronization through using another CUDA kernel is needed. Furthermore, to reduce coding complexity, the shifting operation is performed on one feature image at a time, thus the *shifting_kernel* is called for thrice, in a sequential manner. Considering the large amount of data parallelism which is equal to the total number of pixels attained on shifting one feature image, parallelizing the shifting of 3 feature images would not provide significant speed improvement, owing to the limited GPU resources. A thread in the *shifting_kernel* is responsible for copying d+1 pixel values from some source locations to corresponding destination locations, as mentioned early d=8. The source location in the target feature image is determined by the shifting radius r_s and direction; while the destination location is defined by the global position of the thread, as it can be seen in Figure 5-4. Note that there are d+1 destination matrices for storing the shifted feature image separately. An illustrative sample of the shifting operation performed by a thread is given in Figure 5-5. We apply the mirror image concept to avoid accessing of pixel locations outside image boundary.

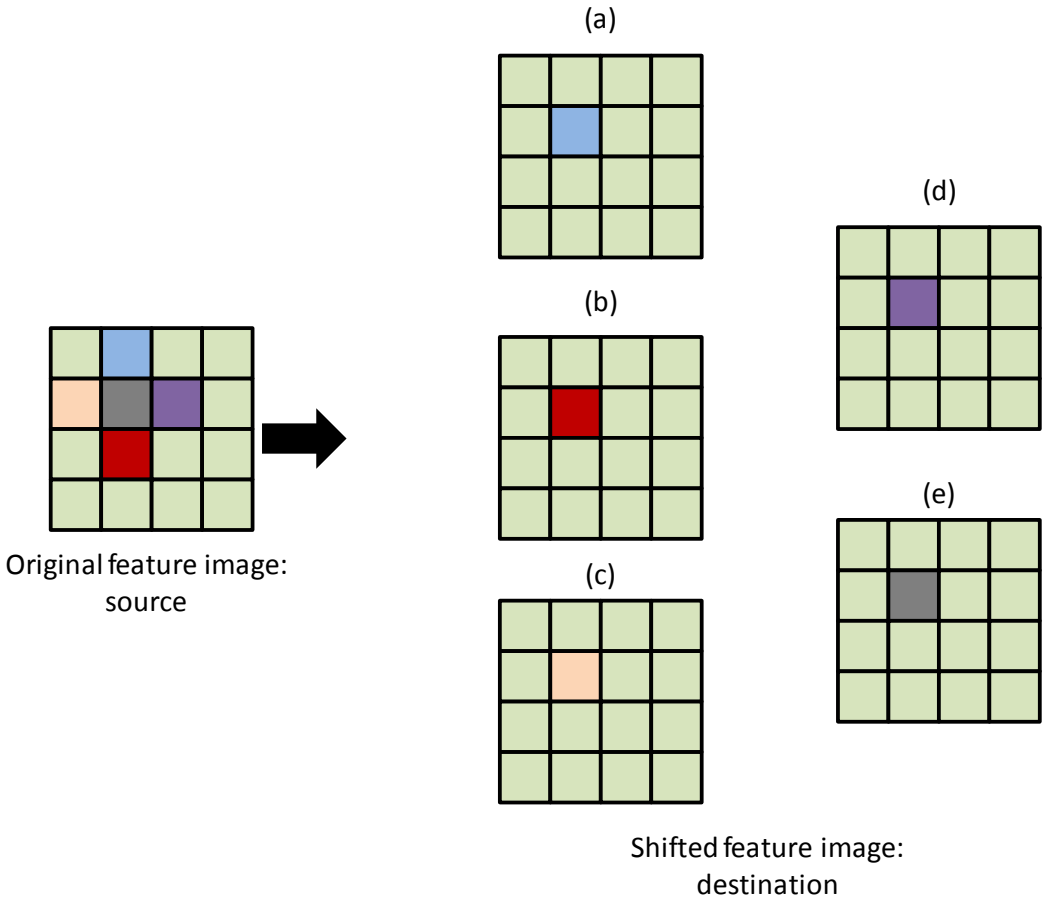

Figure 5-5: An example illustrates the copying action performed by a CUDA thread in *shifting_kernel*, with *r*=1, *d*=4; (a) shift downwards, (b) shift upwards, (c) shift to the left, (d) shift to the right, and (e) no shifting.

In the segmentation CUDA kernel, *segmentation_kernel*, a thread is responsible for comparing a pixel value in pseudo image with the threshold ψ_s, and then assigning a suitable class (either 0 or 1). Both the source and destination pixel location is the same, which is specified by the global position of the thread in the input raw image.

Two additional CUDA kernels, *pre_normalization_kernel* and *normalization_kernel*, and a series of CUBLAS functions are devoted for the PCA modeling task. Since an unfolded matrix is required for PCA modeling as discussed in Section 5.1.2, the 27 images generated from the shifting operation are stored in a 2D matrix with $(X * Y)$ rows and 27 columns. The normalization task can be divided into 27 subtasks, and each subtask is

responsible for normalizing one column in a CUDA block. The *T* threads within the CUDA block iterate through the $(X * Y)$ elements and collect two cumulative values, $\sum_{j=0}^{Y}\sum_{i=0}^{X} I(ij,f)$ and $\sum_{j=0}^{Y}\sum_{i=0}^{X}[I(ij,f) \times I(ij,f)]$, where $I(ij,f)$ denotes the element value at f^{th} column and $(i + j \times X)^{\text{th}}$ rows. The number of iterations, *iter*, required is equal to $\left\lceil \frac{X \times Y}{T} \right\rceil$. This cumulative operation is achieved by a reduction operation performed in shared memory. These two cumulative values are then be used for calculating the mean and standard deviation of a specific feature image (or column), according to Equations (5-4) and (5-5). The computation of normalized values requires both the mean and standard deviation, as given in Equation (5-6). Hence, instead of calling the within-block synchronization at the end of *pre_normalization_kernel,* a separate CUDA kernel denoted as *normalization_kernel* is used for normalization. The thread idling time incurred by within-block synchronization is expected to be as costly as the CUDA kernel call overhead. Essentially, the kernel *pre_normalization_kernel* execute Equations (5-4) and (5-5); and *normalization_kernel* takes care of Equation (5-6).

$$\mu_f = \frac{\sum_{j=0}^{Y}\sum_{i=0}^{X} I(ij,f)}{\sum_{j=0}^{Y}\sum_{i=0}^{X} 1} \quad (5\text{-}4)$$

$$\sigma_f = \frac{\sum_{j=0}^{Y}\sum_{i=0}^{X}[I(ij,f) \times I(ij,f)] - \frac{\left[\sum_{j=0}^{Y}\sum_{i=0}^{X} I(ij,f)\right] \times \left[\sum_{j=0}^{Y}\sum_{i=0}^{X} I(ij,f)\right]}{\sum_{j=0}^{Y}\sum_{i=0}^{X} 1}}{\left(\sum_{j=0}^{Y}\sum_{i=0}^{X} 1\right) - 1} \quad (5\text{-}5)$$

$$I(ij,f)_{norm} = \frac{I(ij,f) - \mu_f}{\sigma_f} \quad (5\text{-}6)$$

Numerous GPU-based PCA algorithms have been proposed in the literature, while some are designed for specific application like (Woo et al., 2013) and (Jošth et al., 2012),

others are generic (Andrecut, 2009; Funatsu & Kuroki, 2010; Zhang & Lim, 2012). A generic algorithm, GPU-NIPALS-PCA, which is proposed by Andrecut (2009) is chosen for two main reasons. In this work, we construct the pseudo image based on the first principal component, so Non-linear iterative partial least squares (NIPALS) algorithm which demonstrates superior efficiency in computing the first few PCA components appears to be an adequate choice. Furthermore, the algorithm provides straightforward and simple implementations by directly using CUBLAS functions. First of all, memory space is allocated for 3 matrices, residual matrix *R*, score matrix *T*, and loading matrix *P*. For determining each principal component, it enters an iterative loop involving several steps, including regression of *R* on *T* which is written to *P* using matrix vector multiplication function *cublasSgemv*; normalization of the *P* using *cublasSscal* and *cublasSnrm2*; regression of *R* on *P* which is written to *T* using *cublasSgemv*; and then computation of the corresponding eigenvalue from *T* using *cublasSnrm2*. These steps will be repeated until the change in eigenvalue in two consecutive iterations is smaller than a user-specified threshold.

CUDA kernel optimization: Except for the joining of 3 feature extraction processes, there is no further merging of CUDA kernels is possible. As discussed before, the 3 main tasks, feature extraction, PCA modeling, segmentation, are inherently sequential. Due to the different task partitioning approaches used in the kernels *shifting_kernel* and *pre_normalization_kernel*, they must be kept in separate kernels. At the same time, "*pre_normalization_kernel* and *normalization_kernel*" are intentionally separated for better tractability. As has already been pointed out, shared memory is used for storing intermediate results which helps to avoid register spillage and promote cooperative action among threads. A major optimization is achieved by data transposition on the shifted images, from $(X * Y) \times 27$ to $27 \times (X * Y)$. The matrix transposition enables

coalesced global memory accessing under below situations, when consecutive threads output results at the end of *shifting_kernel*, access data at the beginning of *pre_normalization_kernel*, and modify data during *normalization_kernel*. This is due to the fact that the CUDA kernels are designed such that consecutive threads access neighboring pixels in $(X * Y)$ direction rather than in the 27 features.

5.3. Case Study: Real-Time Particle Size Estimation in Batch Crystallization of Monosodium Glutamate Monohydrate

The real-time particle estimation capability of our proposed hybrid CPU-GPU-MIA algorithm is tested on a case study of batch crystallization of monosodium glutamate monohydrate (MSG). The experiments were conducted by one of the authors of the article (Sarkar et al., 2009), Ying Zhou. Equipment setup for the batch crystallization is shown in Figure 5-6, where a 500ml flat-bottomed glass vessel fitted with a variable speed stirrer serves as the crystallizer. The crystallization process begins with dissolving 320g of MSG in 400ml deionized water, which is followed by a series of heating for complete dissolution of the particles, and then cooling for crystallization to take place. A graphical representation of the temperature change is summarized in Figure 5-7; the detailed experimental steps are provided in (Sarkar et al., 2009). There are a total of 29898 in situ images captured during the 23 hours crystallization process, using a Particle Vision and Measurement (PVM) probe. The PVM generates images with resolution of 480x640 at the speed of 2 images per second. In this work, the PVM serves as a benchmark for performance evaluation, which means image processing speed of 0.5 second per image is the minimum requirement for real-time application.

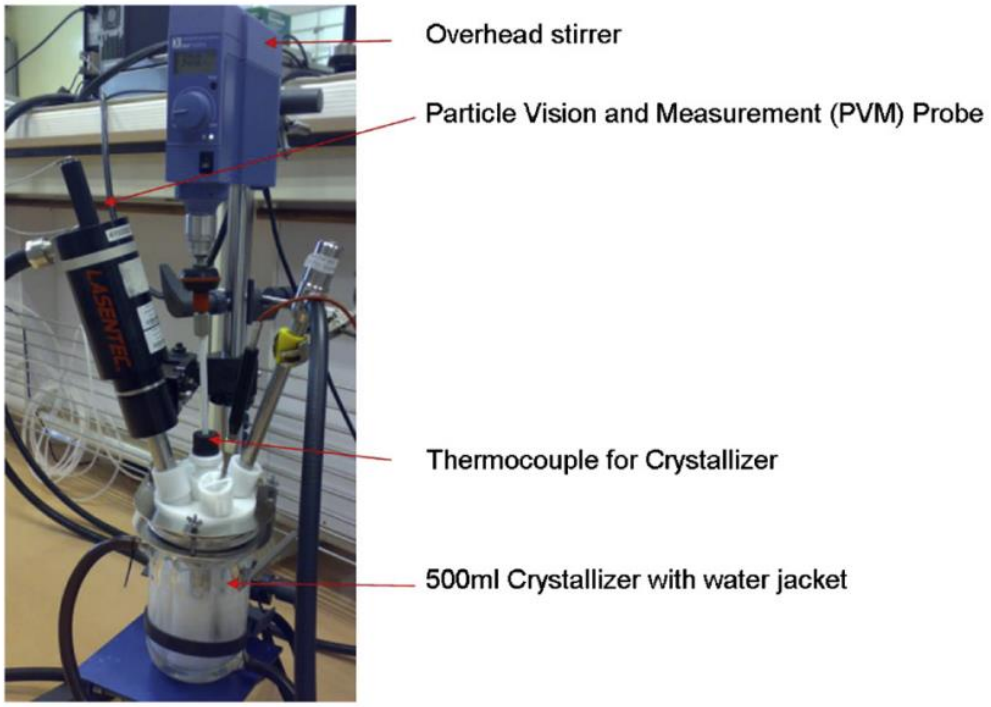

Figure 5-6: Equipment set-up for the batch crystallization case study[10].

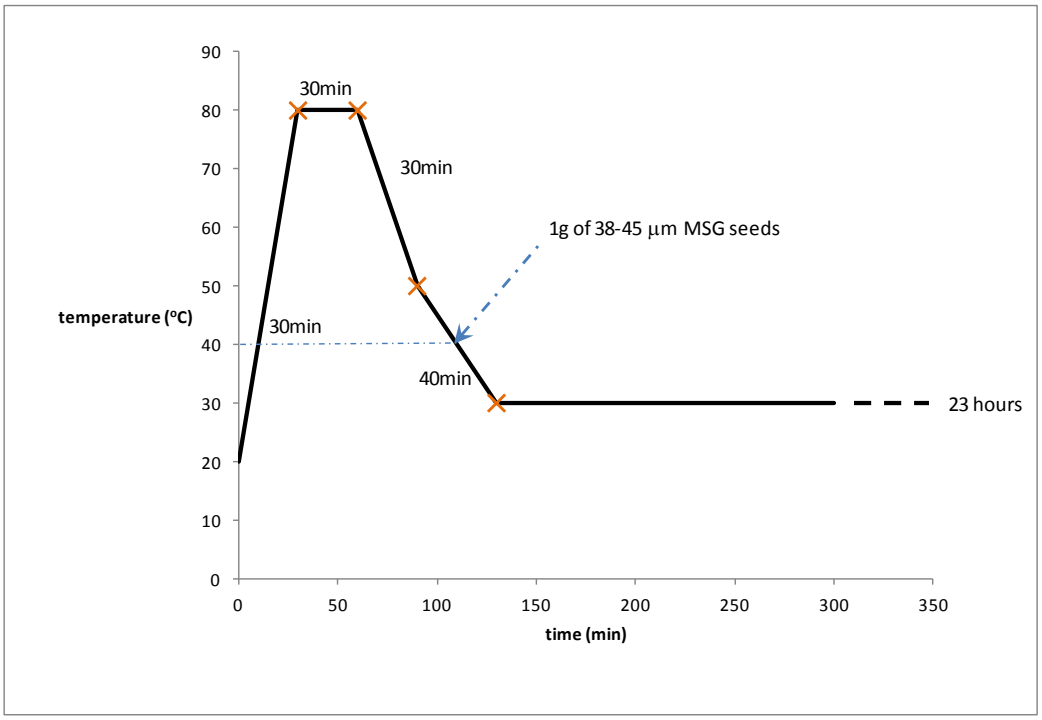

Figure 5-7: Temperature profile of the batch crystallization process.

[10] Courtesy: (Sarkar et al., 2009)

We examine the performance of our approach by comparing the accuracy of particle size estimation and computational efficiency with the original sequential approach. Both the sequential and hybrid MIA algorithms are implemented in Matlab and all the computations are executed on a computing system with 64-bit Intel Xeon CPU with processor speed of 3.20GHz and 12 GB RAM; while an Nvidia Quadro2000 GPU card is used as the co-processor. Standard image processing functionalities such as flood-filling, morphological opening, and object labeling, are performed using Matlab Image Processing Toolbox. The parameter values used in the numerical experiments are given in Table 5-1, while the thread configurations for the 5 CUDA kernels are presented in Table 5-2. All 5 CUDA kernels along with several CUBLAS function calls are wrapped into a MATLAB MEX function. It should be noted that the same set of 29898 images used in the original work (Sarkar et al., 2009) for performance evaluation, is employed in this work.

Table 5-1: Parameter values of MIA

Parameters	Value
Feature selected, f	Range Standard deviation Entropy
Number of PCs retained, N_p	1
Threshold parameter, α	2.5
Pixel area threshold, ψ_a	200
Window size, B	1
Radius of shifting, r_s	1
No. of shifting direction, d	9
Threshold value, ψ_s	$(\propto \times \psi_s) = 2.5 \times 2.7876 = 6.969$

Table 5-2: Thread configurations used for the 5 CUDA kernels

Kernel	Thread configurations	
	Grid size [x,y]	Block size [x,y]
feature_kernel *shifting_kernel* *segmentation_kernel*	$\left[\left\lceil \frac{\text{no. of pixels in x direction}}{32} \right\rceil, \left\lceil \frac{\text{no. of pixels in y direction}}{8} \right\rceil\right]$ =[15,80]	[32,8]
pre_normalization_kernel *normalization_kernel*	$[(f \times d), 0] = [27,0]$	[512,0]

5.3.1. **Accuracy of Particle Size Estimation**

There are two important attributes pertinent to the accuracy of particle sizing, which are the number of particles detected from every image, and the size of each identified particle. Given that the accuracy of original CPU-MIA algorithm has been validated based on manual segmentation (Sarkar et al., 2009), particle size information obtained from CPU-MIA algorithm serves as reliable source for assessing the quality of our proposed hybrid approach. Experimental results show that the number of particles detected in all images is consistent between the CPU-MIA and CPU-GPU-MIA algorithms, except for 4 images as shown in Table 5-3. It accounts for about 0.01% error out of the 29898 images, and it is considered fairly insignificant. Table 5-3 also shows that the discrepancy in quantity of particle is 1 at maximum, which is missing either from the CPU or the hybrid method; one example is given in Figure 5-8.

Table 5-3: Images with different number of particles detected based on the CPU and hybrid algorithms.

Image ID.	No. of particles detected		Difference in no. of particles
	CPU-MIA	CPU-GPU-MIA	
26238	8	9	1
27534	4	3	1
29454	10	9	1
29745	6	5	1

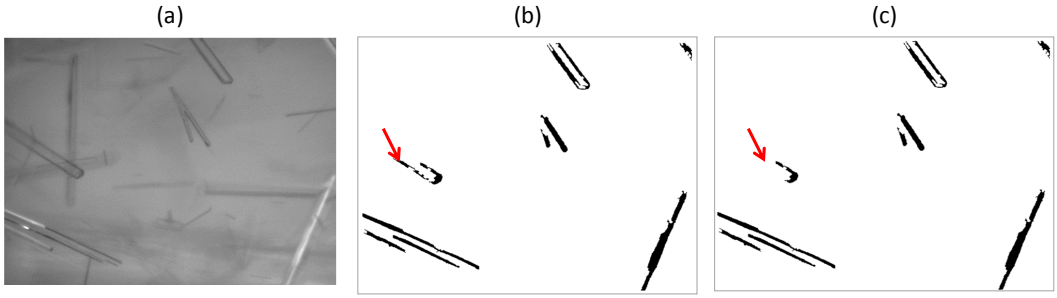

Figure 5-8: Results of applying MIA algorithm to image no. 29454, where an arrow points to the missing particle; (a) grayscale image captured by PVM, (b)(c) post-segmentation image obtained from CPU-MIA and CPU-GPU-MIA, respectively.

Next, we investigate the accuracy of particle size quantification. In MIA approach, particle size is measured in the unit of pixel area i.e. number of pixels, thus comparison can be made based on the metric defined below,

$$\Delta_p = \left| pixel\_area_{p,CPU} - pixel\_area_{p,GPU} \right| \quad (5\text{-}7)$$

where, p is the identity of a common particle detected by the 2 approaches. Excluding those misplaced particles in the 4 images presented in Table 5-3, there are 87541 common particles identified by the 2 approaches. Figure 5-9 plots the particle size difference against the 87541 particles, where the particle identities are given according to the generation order of images. Significantly, the hybrid method provides consistently close particle size measurement to the CPU method for most of the particles under study, excluding several outliers which are found occasionally and randomly throughout the 23 hours crystallization process. In order to provide a clearer view of the overall distribution, the particle size difference is broadly divided into 3

ranges as given in Table 5-4. It can be seen that up to 99.3% of the particles are found having exactly the same size between the 2 approaches (i.e. Δ_p=0), 0.67% particles are slightly different in their size measurement with difference less than 20 pixels, and about 0.005% particles fall into the range of having Δ_p>20 pixels. The largest particle size difference is found at image ID. 16023 in which there are total 4 particles detected. 3 out of 4 particles show zero size difference while the remaining one has 115 pixels missing from the GPU results. A visual comparison of the particle sizes identified with the 2 approaches is presented in Figure 5-10.

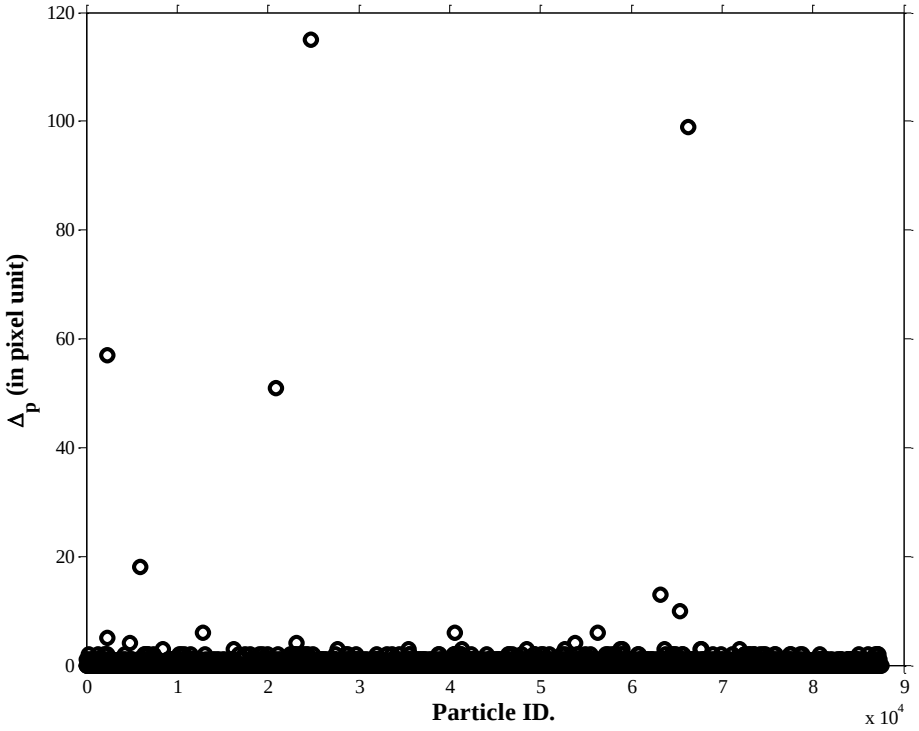

Figure 5-9: Particle size difference between the CPU-MIA and CPU-GPU-MIA algorithms.

Table 5-4: Distribution of particle size difference for the 87541 common particles.

Range of Δ_p	**no. of particles**	**% particles**
0 pixel	86947	99.321%
between 0 and 20 pixels	590	0.674%
>20 pixels	4	0.005%
Total	87541	100%

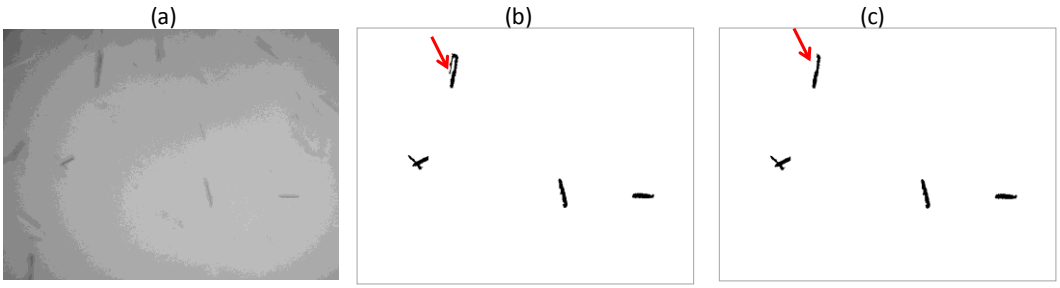

Figure 5-10: Results of applying MIA algorithm to image no. 16023, where an arrow points to the missing particle; (a) grayscale image captured by PVM, (b)(c) post-segmentation image obtained from CPU-MIA and CPU-GPU-MIA, respectively.

5.3.2. Real-Time Computational Efficiency of Hybrid CPU-GPU-MIA Algorithm

The preceding discussion has shown that the proposed hybrid MIA approach promises for accurate particle identification and size estimation, subsequently we will investigate the achieved efficiency enhancement as well as fulfillment of the real-time requirement. To recap, the hybrid MIA algorithm is constructed by parallelizing the most costly tasks, namely feature extraction, PCA modeling, and segmentation on GPU, while the rest is retained in CPU as in the original algorithm. First, the effect of GPU parallelization on total computing time as well as time distribution in MIA algorithm are shown in Figure 5-11 and Figure 5-12, which present the computing time distribution in sequential and hybrid approach, respectively. It can be seen from Figure 5-11 that the 3 GPU-parallelizable tasks (represented as blue diamond symbols) are apparently the computational bottleneck in the original MIA method. By shifting the 3 tasks to GPU, we can see that the computing time of all components (i.e. blue diamonds, hollow red circles and green stars) become more evenly distributed, as illustrated in Figure 5-12.

The change in computing time distribution can be explained by the substantial computing time reduction of the costly GPU-parallelizable tasks. Moreover, the computing time collected from 5 separate numerical experiments is reasonably consistent and stable.

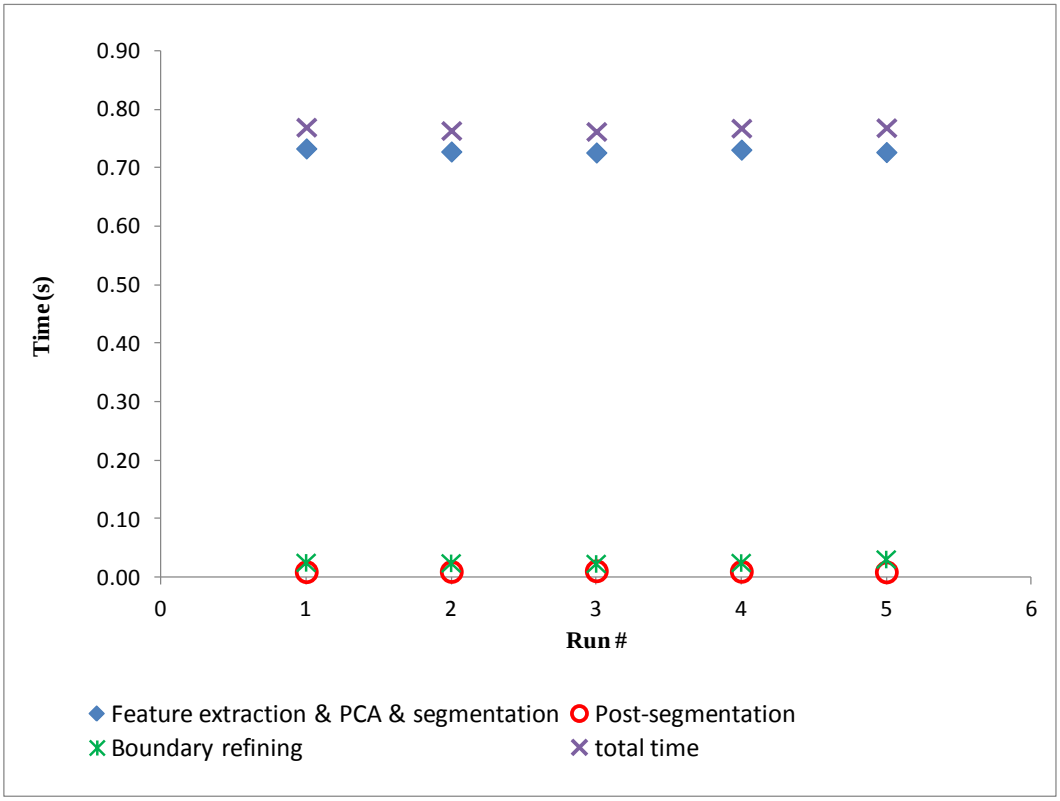

Figure 5-11: Computation time distribution of major steps in CPU-MIA algorithm.

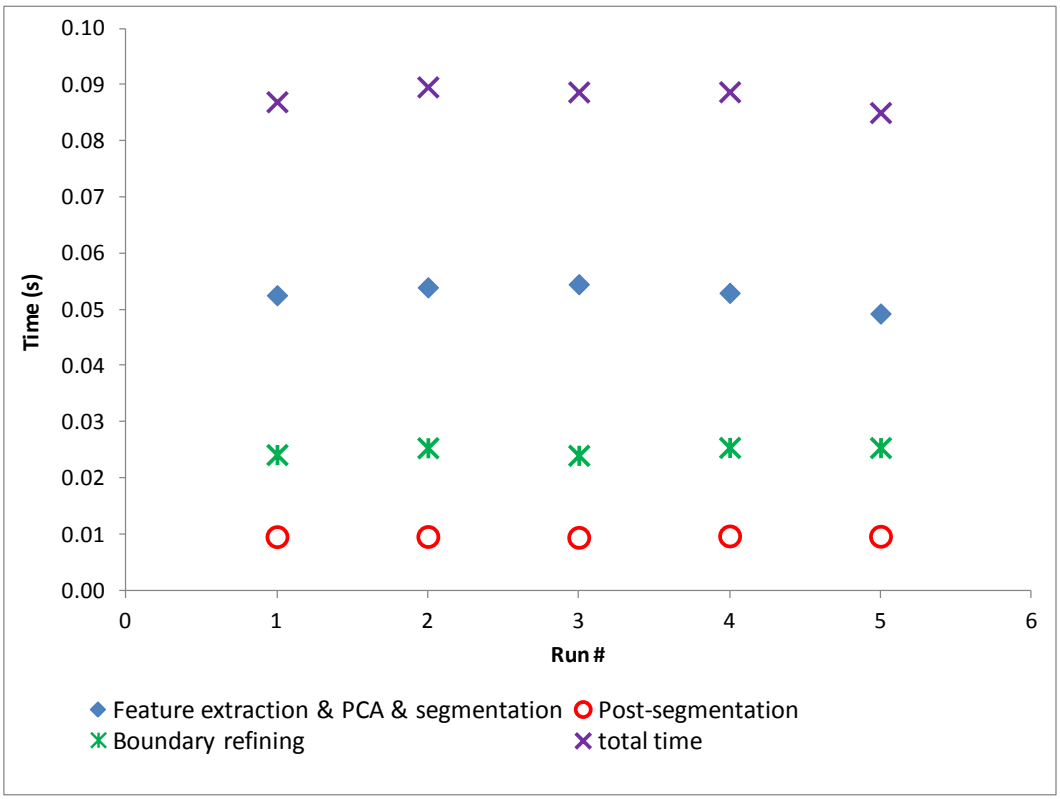

Figure 5-12: Computation time distribution of major steps in CPU-GPU-MIA algorithm.

The speedup ratios computed from the 5 numerical experiments are discussed next. Considering only the 3 GPU-parallelizable tasks, an average of 13.29 speedups are achieved as shown in Table 5-5, and this results in an overall speedups of 8.74, see Table 5-6. The reduced speedup performance can be explained by the inclusion of sequential part. The achieved overall speedups of 8.74 is considerably close to the maximum achievable speedup performance according to Amdahl's law, as follows,

$$Theoretical\ Speedup = \frac{1}{B + \frac{(1-B)}{n}} = \frac{1}{0.09 + \sim 0} = 11.1$$

This value is computed based on 0.09% total time taken by the sequential part (see Figure 5-2), and assuming GPU execution consumes insignificant time. In conclusion, the proposed hybrid CPU-GPU-MIA not only provides accurate particle size estimation, it also offers real-time efficiency by reducing total computing time from 0.77 to 0.09

seconds per image, which is much faster than the image generation speed of PVM at 0.5 seconds per image.

Table 5-5: Computational efficiency enhancement achieved by the CPU-GPU-MIA algorithm on the 3 GPU-parallelizable tasks.

Run#	Speedup
1	13.69
2	13.25
3	13.25
4	13.13
5	13.13
Average	13.29

Table 5-6: Overall computational efficiency enhancement achieved by the CPU-GPU-MIA algorithm.

Run#	Speedup
1	8.86
2	8.53
3	8.60
4	8.66
5	9.05
Average	8.74

5.4. Summary

Image analysis is a powerful process quantification tool, particularly when standard methods like wet chemistry, chromatography are not adequate or inaccurate. Image analysis provides flexibility and effectiveness in quantifying product (or waste) size or shape, based on image processing techniques. As it is required to deal with large image data, real-time process measurement is usually hindered and thus impairing its effectiveness in real-time process monitoring applications. To fully release the value of image analysis techniques, we propose to enhance their computational efficiency using GPU parallelization. In this work, we develop a GPU-based multivariate image analysis algorithm for particle size estimation. In the context of GPU parallel computing, the main contributions of this work include combining multiple feature extractors (or

filtering kernels) in single CUDA kernel to avoid repeated memory access of same input data, and exploring GPU implementation of multi-way PCA. Successful application of the proposed hybrid MIA algorithm has also been demonstrated in a batch crystallization process, in which accurate particle measurement is obtained in real-time. Image processing time is reduced from 0.77 to 0.09 seconds for each image, and this is equivalent to 8.74 speedups.

Chapter 6. GPU-based Data Stream Clustering for Continuous Monitoring of Process Behavior

Clustering is one of the most popular data mining tools which have been successfully applied in many areas. Such popularity is mainly attributed to its simplicity in execution and effectiveness in information extraction. However, it soon becomes ineffective when the data size is too large to be stored in the main memory of the computing system. This can be explained by the fact that traditional clustering algorithms e.g. k-means, PCA-based, hierarchical clustering usually require multiple accesses to the entire dataset, thus expensive data transfer from secondary storage device to main memory is required. It is known that most chemical plants are rich in data but poor in information, as a result of ineffectiveness in handling large dataset. In response to that, a novel research topic called data stream clustering has emerged. Data stream is a data model used for handling transient, large, and rapid data.

In this chapter, we focus on the balanced iterative reducing and clustering using hierarchies (BIRCH) algorithm for two key reasons. The first and most important reason is that BIRCH serves as the fundamental model for deriving many other data stream clustering algorithms like CluTree, DenStream, HPStream, HDDSTREAM, CluStream, and scaleKM. Secondly, we are interested in exploring the use of GPU parallelization for hierarchical clustering algorithm which is inherently lacking of data parallelism. In the interest of developing an accelerated BIRCH approach which provides useful data summaries for various chemical process applications, such as process optimization, scheduling, design, and monitoring, problem size reduction methods like topological decomposition which are strictly problem-specific are not considered in this

work. This chapter is organized as follows. In Section 6.1, we propose an online monitoring scheme which uses BIRCH to detect slow process changes. For an effective online monitoring application, we improve the efficiency of BIRCH through GPU parallelization. The development of the GPU-based BIRCH algorithm is discussed in Section 6.2 wherein kernel design and implementation issues are addressed. The developed hybrid CPU-GPU-BIRCH algorithm is tested on Section 6.3 for clustering quality and process monitoring capability. We first examine the performance of the proposed algorithm using synthetic data and then followed by simulated process data which closely resembles real-world oil and gas production process.

6.1. BIRCH Application to Online Process Monitoring for Gradual Change Detection

BIRCH is particularly suitable for detecting gradual change in process behavior owing to its incremental data summarizing feature. On the one hand, abrupt process change or process fault, e.g. wrong parameters setting due to human fault or power trip, can be detected by focusing on data collected from small time window as the associated change in process measurement is substantial. On the other hand, slow process change occurs at very low speed which may take months or years. Since the change is extremely slow and unnoticeably small, it is necessary to examine much larger data window continuously so as to detect such gradual change. We present an online strategy whereby BIRCH is used to compress process data collected in each window into C clusters, as illustrated in Figure 6-1. C is a user-specified value which is bound by memory space. As C is much smaller than the size of a data window, it allows for more effective data analysis thereafter. Moving windows or sliding windows are commonly used in online data analysis applications, where data is continuously collected by batch and at the same time old data is removed. As discussed in (Adä & Berthold, 2013),

window type can be classified based on two aspects which are the starting location of the window (fixed or sliding) and the width of the window (constant or growing). In this work, a window with sliding starting position and constant width is used for the purpose of comparing consecutive data batches during event detection, as discussed in following paragraphs.

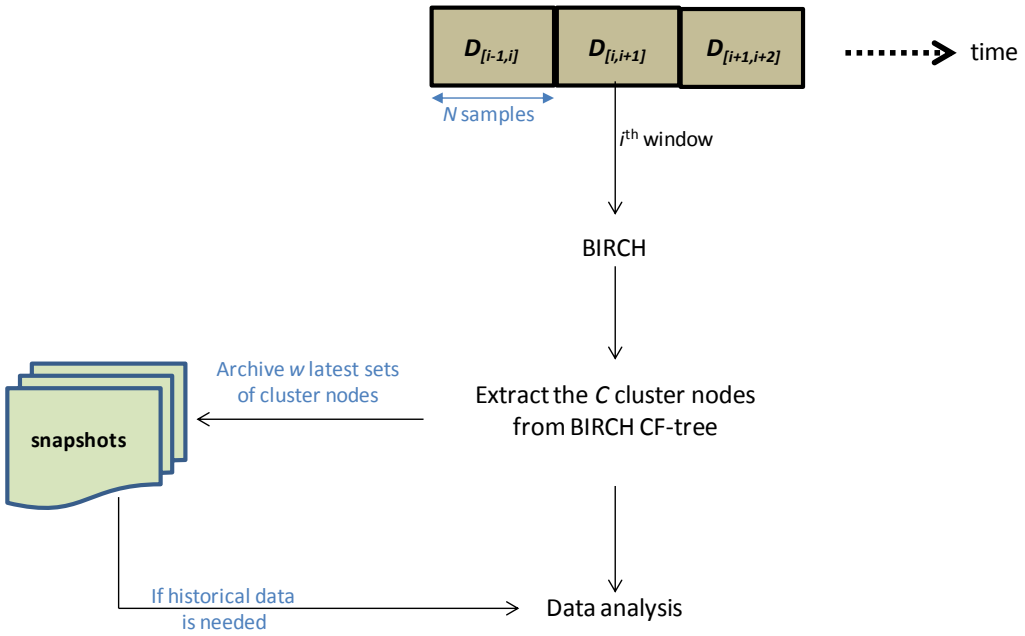

Figure 6-1: Online strategy for detecting gradual process change using BIRCH.

Adä and Berthold (2013) propose an event detection framework wherein events can be identified by tracking the difference in consecutive data windows. Differences are measured based on a dissimilarity function $d(D_i, D_{i-1})$, where D_i refers to data window and *i* is the window identity. However, in cases where data size of *D* is very large, computation of dissimilarity function can be costly and ineffective. Model representations of data which are more concise and noise-free are more appropriate, thus dissimilarity function is redefined as $d(M_i, M_{i-1})$ where *M* denotes the data model. There are 3 basic types of event which are defined as follows,

No event: If the current *d* measurement shows no significant difference from the preceding one, i.e. $d(M_i, M_{i-1}) \approx d(M_{i-1}, M_{i-2})$, there is no sign of event taking place. Note that there are total 4 data windows involved in the computation.

Abrupt event: If the current *d* measurement is significantly larger or smaller than the preceding value, i.e. $d(M_i, M_{i-1}) \gg or \ll d(M_{i-1}, M_{i-2})$, it indicates the existence of abrupt change.

Gradual event: If there is an increasing or decreasing trend of the *d* values across several data windows, it suggests for a gradual event. Mathematically, it is expressed as follows, $d(M_i, M_{i-1}) > d(M_{i-1}, M_{i-2}) > \cdots > d(M_3, M_2) > d(M_1, M_2)$ or (< respectively).

In general, chemical plants involve multiple operating states in order to cater for different product grades and types, or to accommodate for scheduled maintenance or part cleaning; (Méndez & Cerdá, 2003), (Srinivasan et al., 2004) and (Tousain & Bosgra, 2006) show some examples. By using clustering to partition process data collected from different states into separate groups, various process states can be tracked or monitored individually. The dissimilarity function is therefore adjusted to, $d_s(M_{i,s}, M_{i-1,s})$, where *s* represents a particular process state. It is worth mentioning that it is possible for single data window to contain data of different process states, especially if the width of window is not too small. Process state-specific models can be first constructed from identified data clusters, and then used in dissimilarity computation. Alternatively, these data clusters can also be used directly as representative models, in which cluster centers are the model parameters. In this case, dissimilarity between two consecutive windows can be determined by taking difference on the two sets of centroids. It should be noted that the data cluster models are

obtained by applying a standard clustering algorithm (k-means is used in this work) to the *C* cluster nodes acquired from the final BIRCH CF-tree. For more effective clustering (i.e. modeling) during online monitoring, clustering seeds can be first determined offline using historical data based on the number of known normal process states.

Hence, in the interest of detecting gradual change which can deteriorate process performance, it is required to execute the BIRCH algorithm repeatedly on each incoming data window, and the size of data window can be large as explained before. In order to avoid the piling up of data, an efficient BIRCH algorithm is highly favorable. This is particularly important if more complex and costly modeling is needed in dissimilarity determination. In the subsequent section, we develop a GPU-based BIRCH algorithm aiming to enhance the computational efficiency.

6.2. GPU Implementation

As opposed to standard GPU implementation which is targeted at only computationally costly tasks, as illustrated in Chapter 4 and Chapter 5, the entire BIRCH algorithm is executed in GPU. A main reason is that BIRCH operations such as merging, splitting, updating of parent nodes and tree rebuilding are closely related to the status of CF-tree at that point in time, and moving of the CF-tree between CPU and GPU memories is extremely costly and troublesome. Therefore, except for several auxiliary operations like threshold calculation, all operations are executed in GPU. In view of the algorithmic complexity of BIRCH, the discussion on CUDA kernel design, data transferring and optimization is divided into 4 major parts, namely overview of the hybrid algorithm, merging or splitting at cluster nodes, splitting of leaf and nonleaf nodes, and rebuilding of CF tree.

6.2.1. **Overview of the hybrid algorithm**

Figure 6-2 shows the overview of our proposed BIRCH-based online monitoring scheme. Note that CPU is mainly responsible for interacting with database system, building models and calculating dissimilarity value; while GPU is liable for executing BIRCH. In contrast to conventional BIRCH algorithm which processes one sample (or pattern) at a time, we realize data parallelism by handling the incoming samples in batch; β is the notion used for batch size. Given a data window *D*[*i*, *i*+1], we split it into *u* batches with each batch containing β samples except for the first few and the last batch. The initial few batches are intentionally assigned with smaller number of samples due to limited amount of tree nodes at the beginning of CF-tree construction, more explanation will be given later in Section 6.2.2. Whereas in the last batch, it may contain samples lesser than β if the division operation, $\frac{\text{sample size of } D - \text{sample size in initial batches}}{\beta}$, is not exact. In short, the 'GPU-BIRCH' routine represented as a green-outlined box in Figure 6-2 will be called for *u* number of times, and each time a different section of *D*[*i*, *i*+1] is transferred from CPU to GPU global memory.

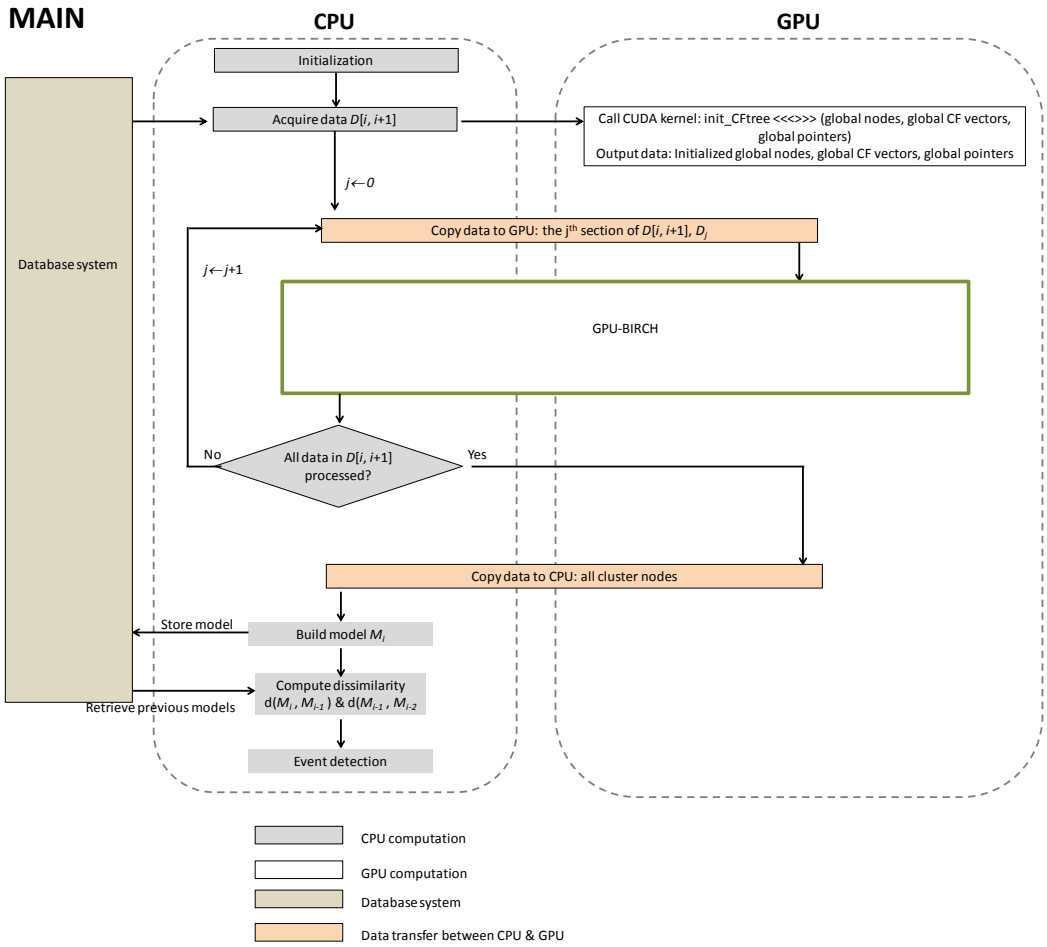

Figure 6-2: Overview of the GPU-based online process monitoring scheme, showing the relationships among database system, CPU and GPU.

Another key difference to conventional BIRCH algorithm is that tree nodes are pre-allocated in GPU-based BIRCH, instead of dynamical allocation. This can be done by translating the user-specified maximum memory space for BIRCH to number of nodes *M*' allowable in the tree. Given that sample dimensionality and page size *P* are known, the size of single node and thus *M*' can be computed. The tree nodes need to be stored in GPU global memory for several reasons, and important ones include the writability requirement and sustained lifetime throughout all kernel calls. As will be discussed in later sections, *LS* and *SS* elements of CF vector and multiple nodes of the same type are always accessed simultaneously by a CUDA thread warp. Hence, for coalesced memory access, same type of nodes like cluster nodes, leaf nodes and nonleaf nodes are

allocated together. Also, CF vectors of the same type of nodes are allocated collectively in the same memory space, as shown in Figure 6-3. By doing so, it is possible to have coalesced memory access through ensuring dimensionality (or elements in *LS* or *SS*) is multiples of warp size i.e. 32. Using CUDA C++ programming, the 3 types of tree nodes are inherited classes from the same base class; function members of the base class include the *N* component of CF vector, child pointers, parent pointers, and tree level.

It can also be seen from Figure 6-3 that there are two levels of pointers used to establish the relationships of nodes and CF vectors. At the outermost level, a pointer array serves as an interface for node manipulation. It provides flexibility in situations where sorting or compacting of nodes is needed, as it can be done without physically moving the nodes around, and thus saving time for copying and deleting. Note that 3 separate global pointer arrays are used for manipulating the cluster, leaf, and nonleaf nodes. Within each node, there is also a pointer connecting it to the associated CF vector. As shown in Figure 6-2, a CUDA kernel named *init_CFtree* is called at the beginning of every data window. The kernel is responsible for setting up all the pointers and also constructs a minimal CF tree which consists of only one cluster node, one leaf node and one nonleaf node. The 3 nodes are related through parent-child relationship and they incorporate the very first sample of each data window.

In this work, a standard parallelization strategy is employed which maps CUDA blocks to samples or nodes, and maps CUDA threads to the elements of CF vector; unless otherwise stated. For instance, in *init_CFtree* kernel, one block is assigned to a node in which one thread is used to perform the pointer assignment, and all threads perform zeroing of elements in the CF vector.

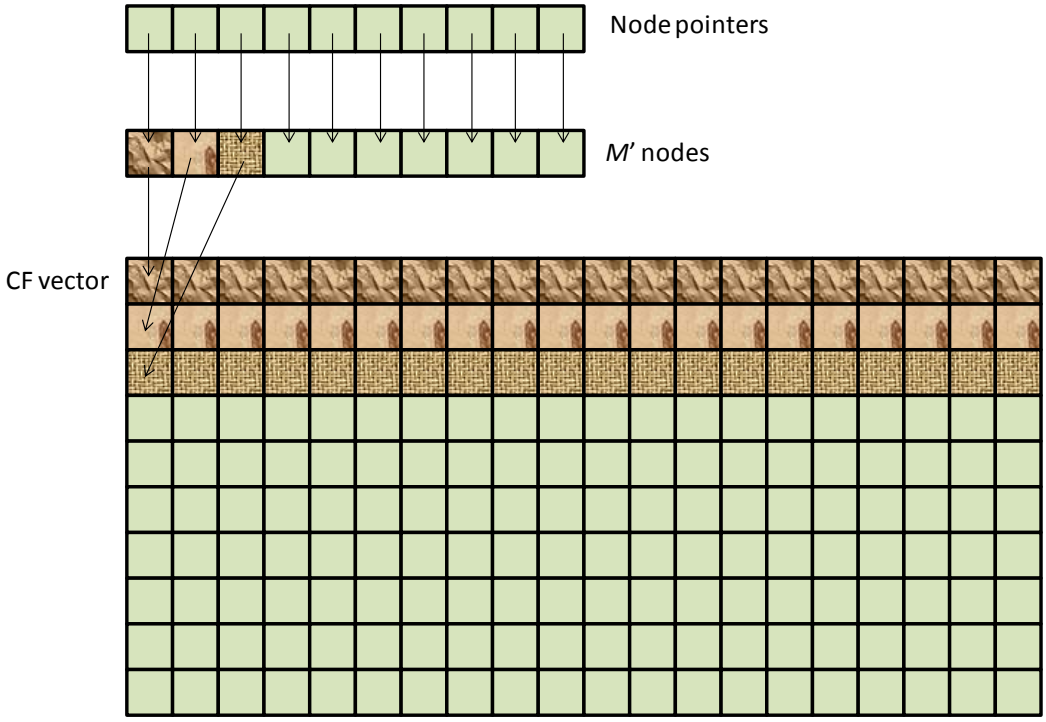

Figure 6-3: Pointer assignment performed in *init_CFtree* CUDA kernel.

Next, we examine the 'GPU-BIRCH' routine which involves a number of CUDA kernels and CPU operations, as depicted in Figure 6-4. Given a data batch D_j extracted from the current data window, it needs to firstly identify the nearest cluster nodes for each samples. The CUDA kernel, *search_nearestCluster*, parallelizes this searching operation over the samples in D_j based on the standard parallelization strategy. Thread-level parallelization is mainly achieved at distance computation, whereby CUDA threads access contiguous elements in CF vectors of both sample and candidate node. It is important to highlight that such CUDA kernel design is mostly beneficial for problems with sufficiently high dimensionality which is the center of our interest. If problem dimensionality is less than the size of a warp which is the smallest block size, some threads will idle and resources are wasted.

There are several output variables used to store computing results of *search_nearestCluster*, which include *action*, $C_{nearest}$, and L_{parent}. The length of these

variables is equal to data batch size, so that the action, nearest cluster, and the parent of the nearest cluster obtained from the s^{th} sample can be stored in the s^{th} element of the variables, respectively. Note that *action* is an integer array using 0 and 1 to denote merging and splitting action at cluster node, while $C_{nearest}$, and L_{parent} are pointer address arrays. Memory space allocated for these variables can be used for storing other intermediate results when their jobs have accomplished.

Following the CUDA kernel *search_nearestCluster*, a counting function from Thrust library is called to determine the number of samples requiring merging and cluster splitting, based on 0's and 1's counts in *action*. Since cluster splitting is only allowed if there is sufficient amount of unused nodes available, it is necessary to ensure that the amount of new clusters required is less than the available nodes. If this is the case, a sub-routine called 'merging or splitting at cluster nodes' will be executed, otherwise 'rebuilding of CF tree' routine will be called to free some of the used nodes by using a larger *T*. Supposing 'merging or splitting at cluster nodes' is chosen, and the number of new clusters created at a particular leaf parent (pointed to by L_{parent}) exceeds the width of the leaf node, splitting at higher level will be required and the sub-routine 'splitting of leaf and nonleaf nodes' will be summoned this time. On the contrary, if all newly created clusters can be absorbed by the leaf parents, then the next step is to update the CF values of higher level nodes to reflect the adding of new samples.

The CUDA kernel, *update_parent*, is designed to update the parent nodes by iterative through the levels from leaf to root node, i.e. one level at a time. Similar to conventional BIRCH algorithm, CF values of upper level nodes are dependent on their descendant nodes, thus synchronization is required at every level. Unlike the original algorithm, the CUDA kernel *update_parent* updates all nodes at a particular level

simultaneously, regardless if their descendant nodes undergo any changes in the 'merging or splitting at cluster nodes' routine. This is due to the reason that tracking of the affected nodes is both time and space consuming, in addition, updating of unaffected nodes in GPU is free though it is unnecessary. The 'GPU-BIRCH' routine will be repeated until all samples in the current window $D[i, i+1]$ are accounted for, as illustrated in Figure 6-2. Upon the last call of 'GPU-BIRCH', the average values of CF vector (i.e. $\frac{LS}{N}$) of every cluster node will be extracted and copied to CPU for model construction and dissimilarity computation.

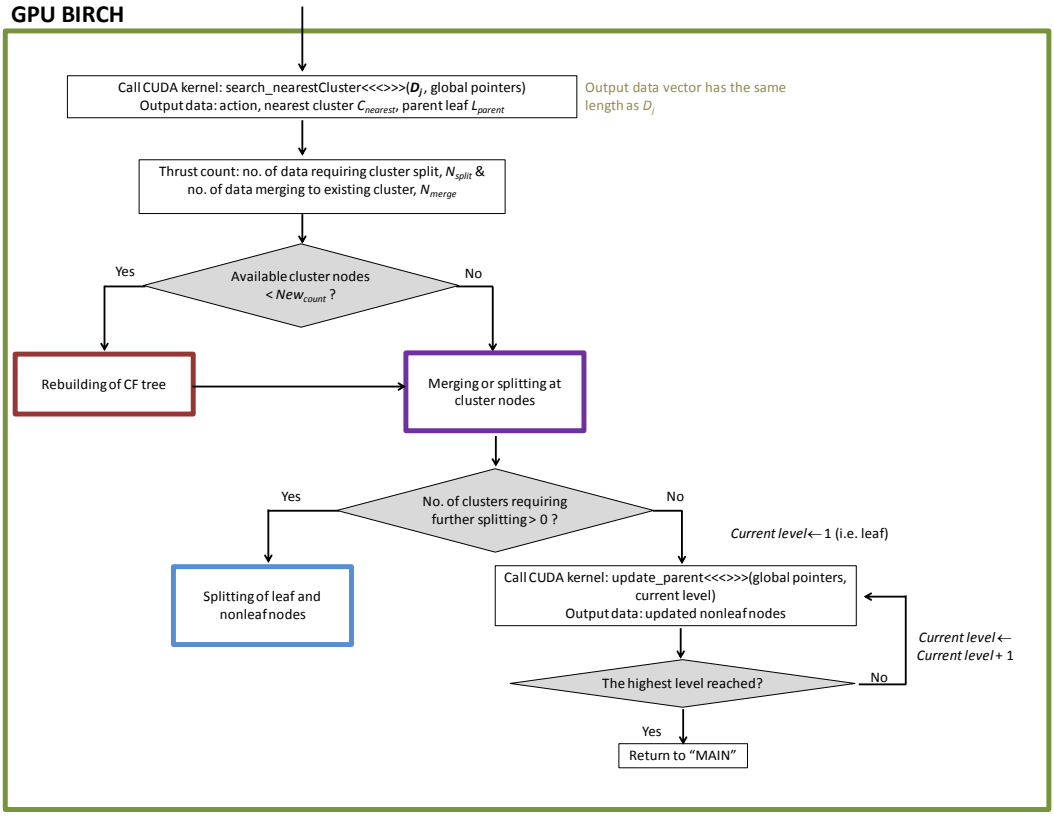

Figure 6-4: Overview of BIRCH implementation in GPU.

6.2.2. **Merging or splitting at cluster nodes**

In this section, we provide the details of the sub-routine 'merging or splitting at cluster nodes' which is represented as a purple box in Figure 6-4. As presented in Figure 6-5, it begins with 3 sorting operations whereby samples merging to the same cluster or creating new cluster at the same leaf node are located together. A variable *sorting_ID* is used to keep track the sample sequence. The first sorting attempts to separate samples requiring merging from those demand for new cluster, using *action* as the sorting keys. To ensure consistency, $C_{nearest}$ and L_{parent} are subsequently re-arranged according to the sorted sample sequence indicated by *sorting_ID*. The remaining two sorting operations are performed on $C_{nearest}$ and L_{parent}, respectively. Since samples to be merged to a particular cluster (or creating new cluster at the same leaf) will have same memory address in the $C_{nearest}$ (or L_{parent}), sorting of the memory addresses in $C_{nearest}$ (or L_{parent}) can partition samples into groups with same nearest cluster (or same parent leaf node). An illustrative sample is given in Figure 6-6.

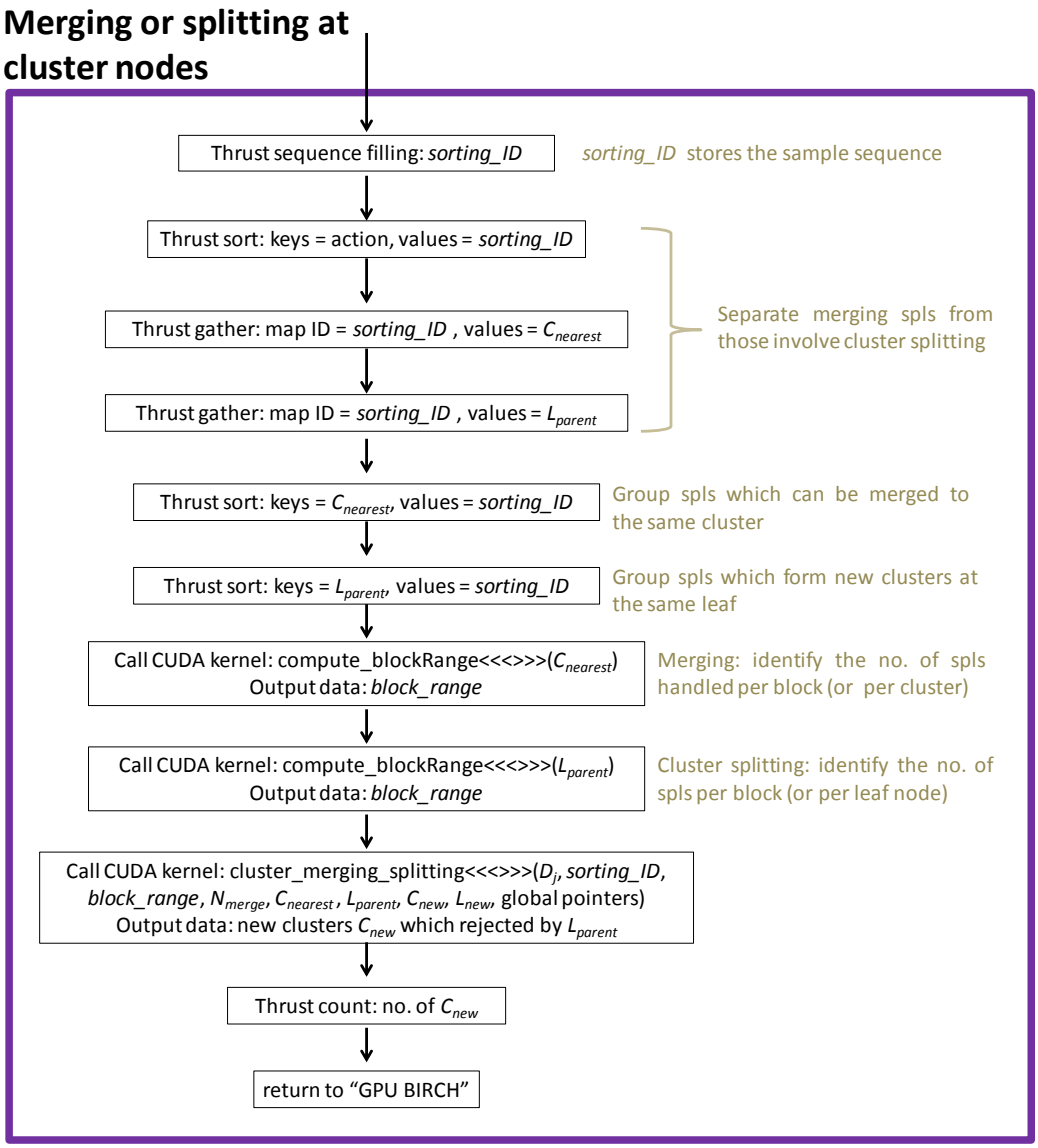

Figure 6-5: Procedure of sub-routine 'merging or splitting at cluster nodes'.

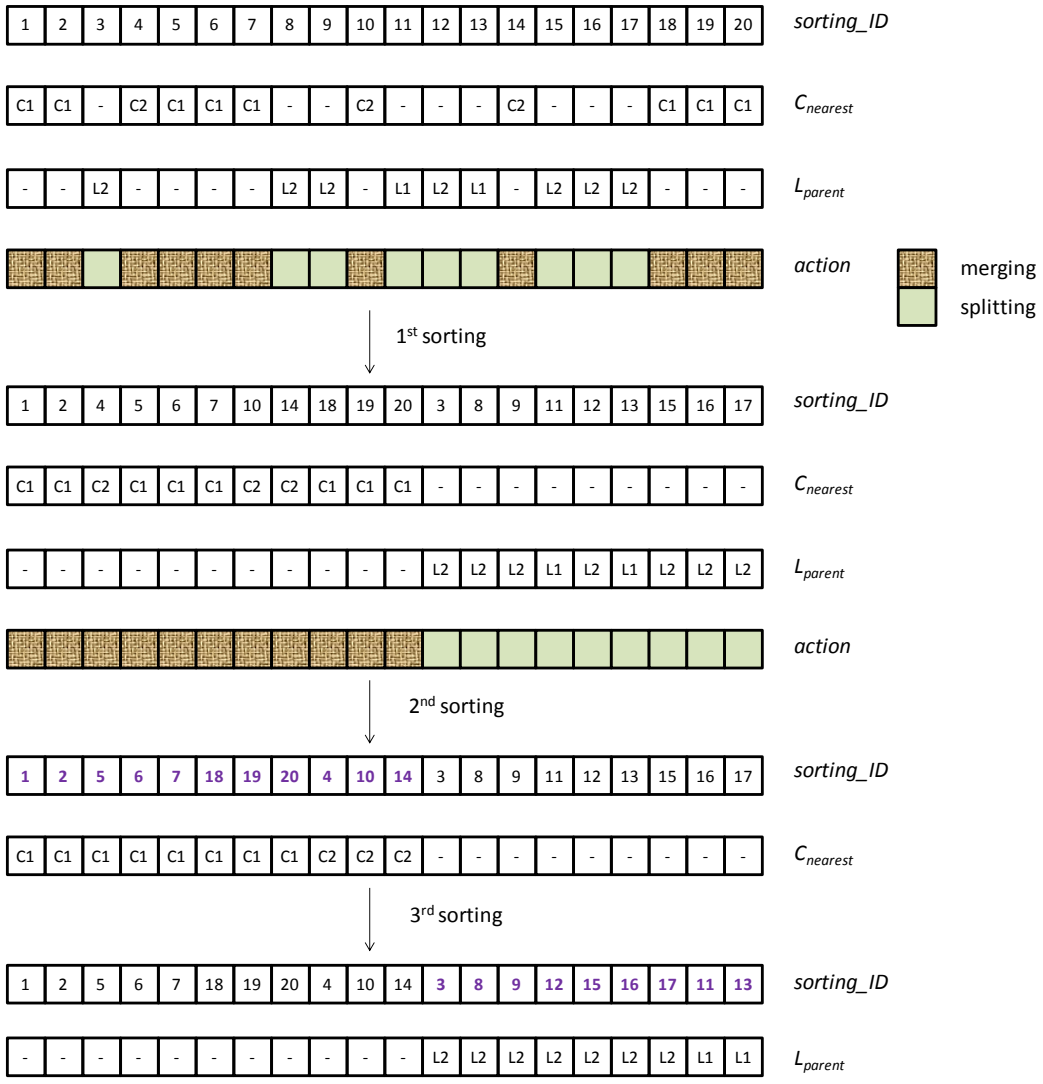

Figure 6-6: A simple example demonstrating the 3 sorting operations involved in sub-routine 'merging or splitting at cluster nodes'.

According to the standard parallelization strategy mentioned in Section 6.2.1, CUDA blocks are mapped to unique cluster nodes (or unique leaf nodes) for merging (or splitting) operations. Samples in the same cluster node (or leaf node) are examined sequentially in the CUDA block, in which CUDA threads are assigned to handle different elements of CF vectors of either sample or node. Take the example given in Figure 6-6, as there are 2 unique cluster nodes (i.e. C1 and C2), 2 CUDA blocks are needed to perform the merging; similarly, another 2 CUDA blocks are used for splitting at the 2 unique leaf nodes, L1 and L2. Since the number of samples managed by each block is

different, a CUDA kernel named *compute_blockRange* is used to determine the sample range. A thread is responsible to retrieve and compare 2 neighboring elements in $C_{nearest}$ and L_{parent}, and write its location to output variable *block_range* if the 2 neighboring elements are different in values, as illustrated in Figure 6-7. Although this CUDA kernel functions similarly to the scanning function available from Thrust library, the latter is not allowed to take more than one input; in this case the 2 inputs $C_{nearest}$ and L_{parent} are scanned simultaneously.

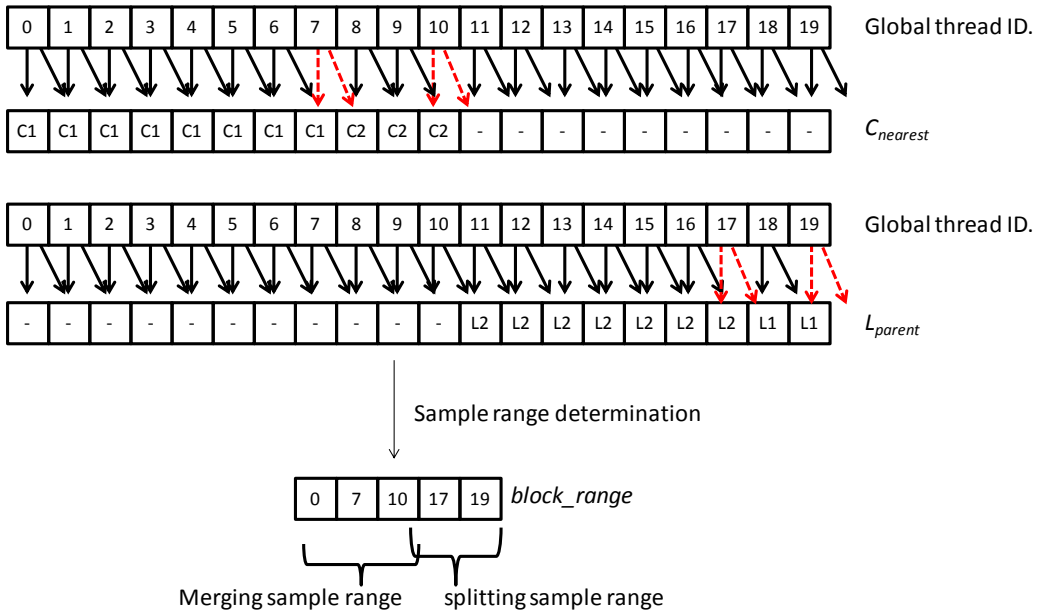

Figure 6-7: Sample range determination using CUDA kernel *compute_blockRange*; using the same data from the example given in Figure 6-6.

The last CUDA kernel to be discussed in this sub-routine is *cluster_merging_splitting* which carries out the actual merging and splitting operation at cluster node. It requires several input variables which include the samples, sorted sample sequence *sorting_ID*, sample range, number of sample groups for merging N_{merge}, $C_{nearest}$, L_{parent}, global cluster pointers, and 2 output variables C_{new} and L_{new} for storing newly created clusters and parents. The global cluster pointers are required for getting the address of unoccupied cluster nodes. To prevent multiple CUDA blocks

compete for the same unoccupied cluster node, each block is restricted to a range of elements in the global array. This range can be determined by the sample range indicated in *block_range* and also location of the first unoccupied node in the global pointer array, as illustrated in Figure 6-8. This example also shows the importance of global node compaction.

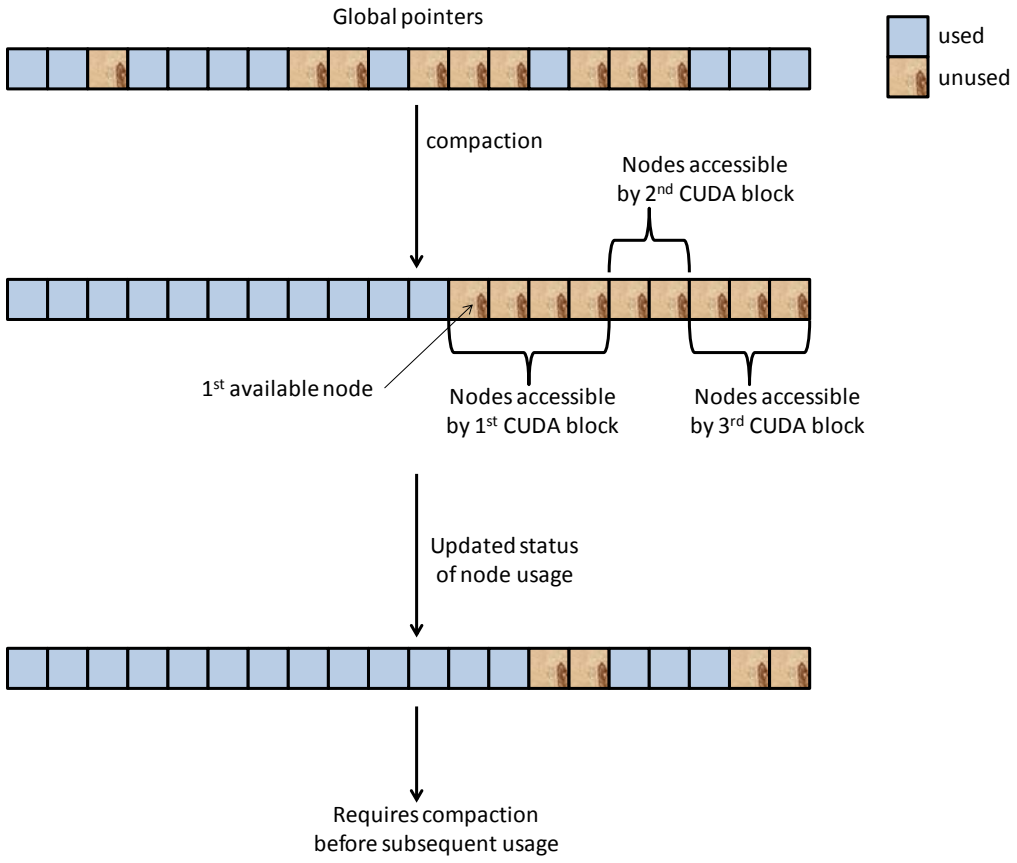

Figure 6-8: Example illustrating the usage and compaction of global pointers.

Given N_{merge}, individual CUDA block is able to find out if it is related to the merging or splitting job, as merging samples are already sorted to the front. Merging of samples to a unique cluster node is achieved by adding samples to the CF vector of the cluster, where the samples identity can be obtained from *block_range* and *sorting_ID*. Similarly, splitting at a cluster node essentially involves creating new cluster at a unique parent leaf node. In a CUDA block, one cluster is created at a time whereby the samples

are examined sequentially to be added into this cluster; new cluster address is obtained from the global cluster pointers. Note that thread-level parallelism is achieved during distance computation and adding of sample to cluster, by mapping threads to variables in sample data or CF vector. If there is empty child lot at the parent leaf node, the parent will absorb the new cluster as a child. Otherwise, a pointer will be generated in C_{new} and L_{new} to store the address of this cluster node and its original leaf node parent, for further processing in later stage. On completion of *cluster_merging_splitting*, the number of new cluster nodes without parent can be counted from C_{new}, and it proceeds to the sub-routine 'splitting of leaf and nonleaf nodes' if the count is not zero.

6.2.3. **Splitting of leaf and nonleaf nodes**

The sub-routine 'splitting of leaf and nonleaf nodes', represented as a blue box in Figure 6-4, involves splitting of nodes at leaf and nonleaf levels. Like conventional BIRCH, the splitting task starts at the leaf level after receiving cluster nodes from preceding sub-routine 'merging or splitting at cluster nodes'. At leaf level, cluster nodes are the child nodes (indicated by $Child_{new}$) which require splitting of their original parent leaf node (indicated by $Parent_{new}$), so that more space is created by having additional parent leaf node. In cases where the number of newly created leaf nodes is more than the available child slot at their immediate nonleaf parent node, higher level node splitting is required. Such splitting may propagate all the way up the CF-tree until there is no more unhandled child node. Note that if it requires splitting at the root node level, the tree height is increased one.

As it can be seen from Figure 6-9, at the beginning of every level of splitting, new child nodes $Child_{new}$ are segregated such that those belong to the same parent are placed together, using sequencing, sorting and gathering functions from Thrust library.

The CUDA kernel *compute_blockRange* is then executed to determine the number of child nodes associated to each unique parent in $Parent_{new}$. As both determination of the farthest pair of child nodes (serve as seeds) and child node re-distribution have computational complexity of $O(C'^2)$, where *C'* is the number of child nodes under consideration, node splitting will become extremely costly when *C'* is large. In order to avoid severe workload unbalance during splitting, we suggest identifying the most crowded block whose *C'* exceeding a user-specified limit α, and performing the distance computation in a separate kernel called *compute_child2child_dist*. Note that the maximum *C'* can be determined from *block_range*, and at most only one block will have its distance pre-calculated due to memory space limitation. In *compute_child2child_dist*, efficient computation can be achieved by mapping *C'* CUDA blocks to *C'* child nodes, and mapping CUDA threads to elements of CF vectors during distance computation. The calculated distances are stored in the variable, *dist_matrix*, which resides in global memory.

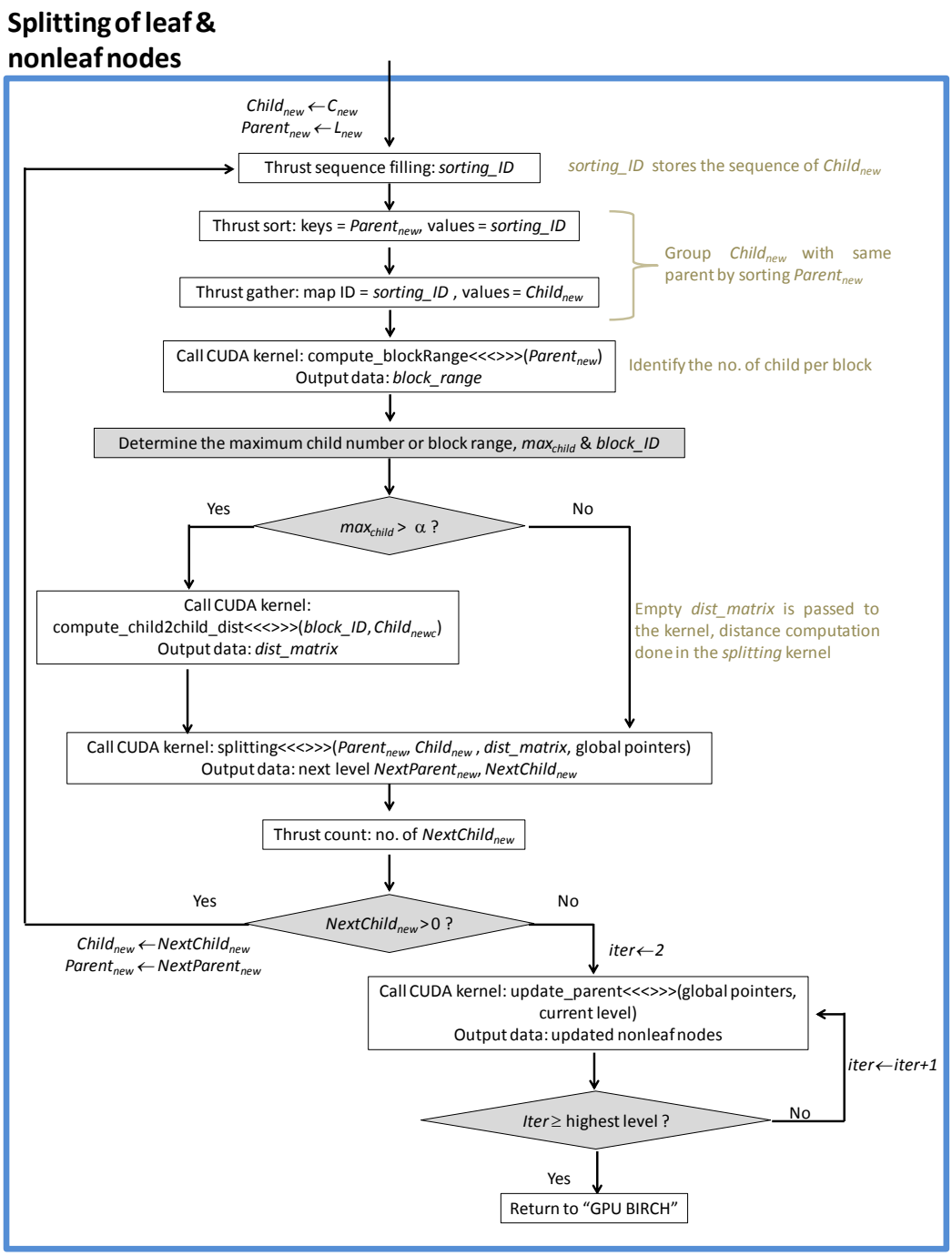

Figure 6-9: Procedure of sub-routine 'splitting of leaf and nonleaf nodes'.

The actual splitting job is performed in a CUDA kernel called *splitting*, which consists of 2 key operations including determination of farthest seeds and redistribution, where a CUDA block is assigned to a unique parent node in $Parent_{new}$. In contrast to conventional BIRCH method, there may be more than one child node needs

to be added to the parent assigned to a CUDA block. As a consequence, the 2 key operations may be repeated for several times until all child nodes are accounted for. Supposing a CUDA block has the child node-to-child node distances pre-computed in *compute_child2child_dist*, it can perform seed determination and child re-distribution directly based on *dist_matrix*. Otherwise, it is required to compute the distance during execution of *splitting* kernel. Figure 6-10 presents a simple example of node splitting occurring at leaf level. It should be noted that the creation of new nodes is done through the global pointers, as depicted in Figure 6-8. The splitting procedure will be repeated until the $NextChild_{new}$ is found empty. It is then required to update the CF vectors of all higher level nodes to reflect the adding of new samples or new child nodes, before returning to the 'GPU-BIRCH' routine.

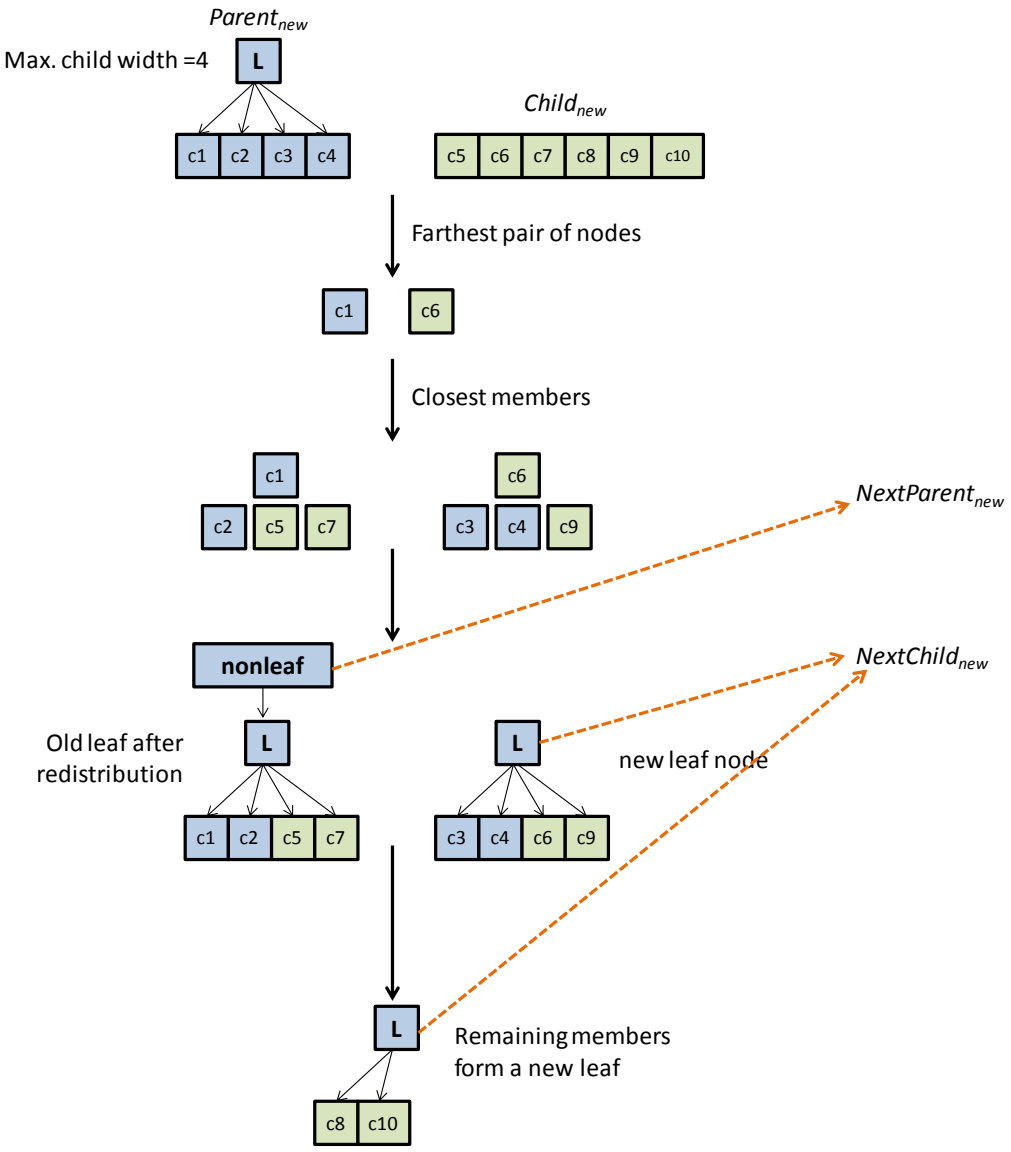

Figure 6-10: An illustrative sample of node splitting and redistribution at leaf level, in GPU-based BIRCH algorithm.

6.2.4. **Rebuilding of CF tree**

Another sub-routine named 'rebuilding of CF tree' is used to compress CF-tree by using a larger *T*; it is represented as a red box in Figure 6-4. In conventional BIRCH, the target for CF-tree rebuilding is to free up one cluster node, as there is only one sample being handled at one time. In our proposed GPU approach, β samples are dealt with simultaneously, so it is important to first determine the number of new cluster nodes required. Upon completion of the CUDA kernel *search_nearestCluster*, the

number of samples requiring splitting N_{split} will be known. Here, we assume a maximum number of new cluster nodes to be created, which is equal to N_{split}. In other words, it is assumed that all candidate samples are far from each other such that every newly created cluster contains only single sample. Setting a maximum requirement is necessary as the CF-tree rebuilding is executed prior to the sub-routine 'merging or splitting at cluster nodes' and the actual number of new clusters needed is still unknown. This strategy avoids an additional execution of *cluster_merging_splitting*, but it also risks for an unnecessary high T as will be discussed in later Section 6.3. Based on this maximum requirement, the rebuilding operation is repeated until the target is met.

As shown in Figure 6-11, the first part of tree rebuilding is the determination of $D_{\min}$ which is defined as the distance between two closest child nodes in the most crowded leaf node. In order to find the most crowded leaf node, a CUDA kernel called *extract_N_from_leaf* is employed, in which T' CUDA threads (where, $T' \geq L'$) are used to copy the number of samples N and memory address of the L' leaf nodes to variables N_{leaf} and L_{ptr}, respectively. In this case, arbitrary thread configuration is acceptable. The most crowded leaf node can then be pushed to the front by carrying out sorting on L_{ptr} using N_{leaf} as keys. Determination of $D_{\min}$ is executed in another CUDA kernel, *compute_Dmin*, by assigning single CUDA block to examine the first element in L_{ptr}. Thread-level parallelization is achieved by mapping CUDA threads to elements of CF vectors for calculating inter-cluster distances among all possible pairs of the child nodes. The minimum distance found will be output to the variable $D_{\min}$ which is subsequently copied to CPU for calculating new threshold T.

Given an increased T value, some of the cluster nodes can be merged and therefore freed for new samples in N_{split}. To account for the possibility of merging cluster

nodes located at different leaf nodes, we propose mapping CUDA blocks to nonleaf nodes at level 2, which are immediately above the leaf node. Since single global pointer array is shared by nonleaf nodes at all levels (from level 2 to root node level), we launch a number of CUDA blocks that is equal to the total number of occupied nonleaf nodes while executing the CUDA kernel, *rebuild_CFtree*. At the beginning of the kernel, a CUDA block will terminate its operations if the associated nonleaf node is not at level 2. Otherwise, it will iterate through all the child cluster nodes, one at a time. For each cluster node $c1$, it determines the nearest cluster node $c2$ which is not necessary under the same parent node. In cases where the distance between $c1$ and $c2$ is smaller than the new T value, the two clusters will be merged, and the corresponding parent leaf nodes will also be updated. Obviously, if a cluster has already been merged previously, it will not be examined again in the remaining iterations.

Following the cluster merging operation in *rebuild_CFtree*, there are 2 CUDA kernels which are *remove_emptyLeaf* and *remove_emptyNonLeaf* used to remove empty nodes at leaf and nonleaf levels, respectively. In these 2 kernels, one CUDA block is mapped to one leaf or nonleaf node, in which its corresponding child nodes are examined sequentially. If any child node is found empty, the last available child will be moved to this position. In doing so, the child pointer array is compacted for future usage. As *remove_emptyNonLeaf* is performed on one level nonleaf nodes at one time, a CUDA block will return immediately if it finds that the assigned node is not at the target level.

It should be noted that the CUDA kernel *rebuild_CFtree* is designed in a way that it ignores any merging opportunity at level higher than 2, and this may lead to the formation of long single branches in the tree; a long branch is formed when a higher

level nonleaf node consists of single child at all lower levels. To allow merging at higher levels, a CUDA kernel called *merge_single_branch* is employed. For more efficient computation, the existence of long single branches is first identified in the kernel *remove_emptyNonLeaf* where multiple nonleaf nodes can be examined simultaneously. In view of the rarity of long single branch, single CUDA block is launched for *merge_single_branch* in which one identified branch is examined at a time. Despite of the limited parallelism, it can avoid complication of parent node tracking which is needed to prevent race condition among CUDA blocks. Like *rebuild_CFtree*, this kernel also seeks to merge any 2 clusters whose inter-cluster distance is smaller than *T*. The only difference is that *merge_single_branch* traverses from higher level, while *rebuild_CFtree* starts from level 2.

At this stage, the CUDA kernel *search_nearestCluster* needs to be executed once more to re-calculate the N_{split} based on the compressed CF-tree and updated *T*. Assuming the newly computed N_{split} is still larger than the available node, the entire CF-tree rebuilding procedure will be repeated. Otherwise, it will return to the 'GPU-BIRCH' routine.

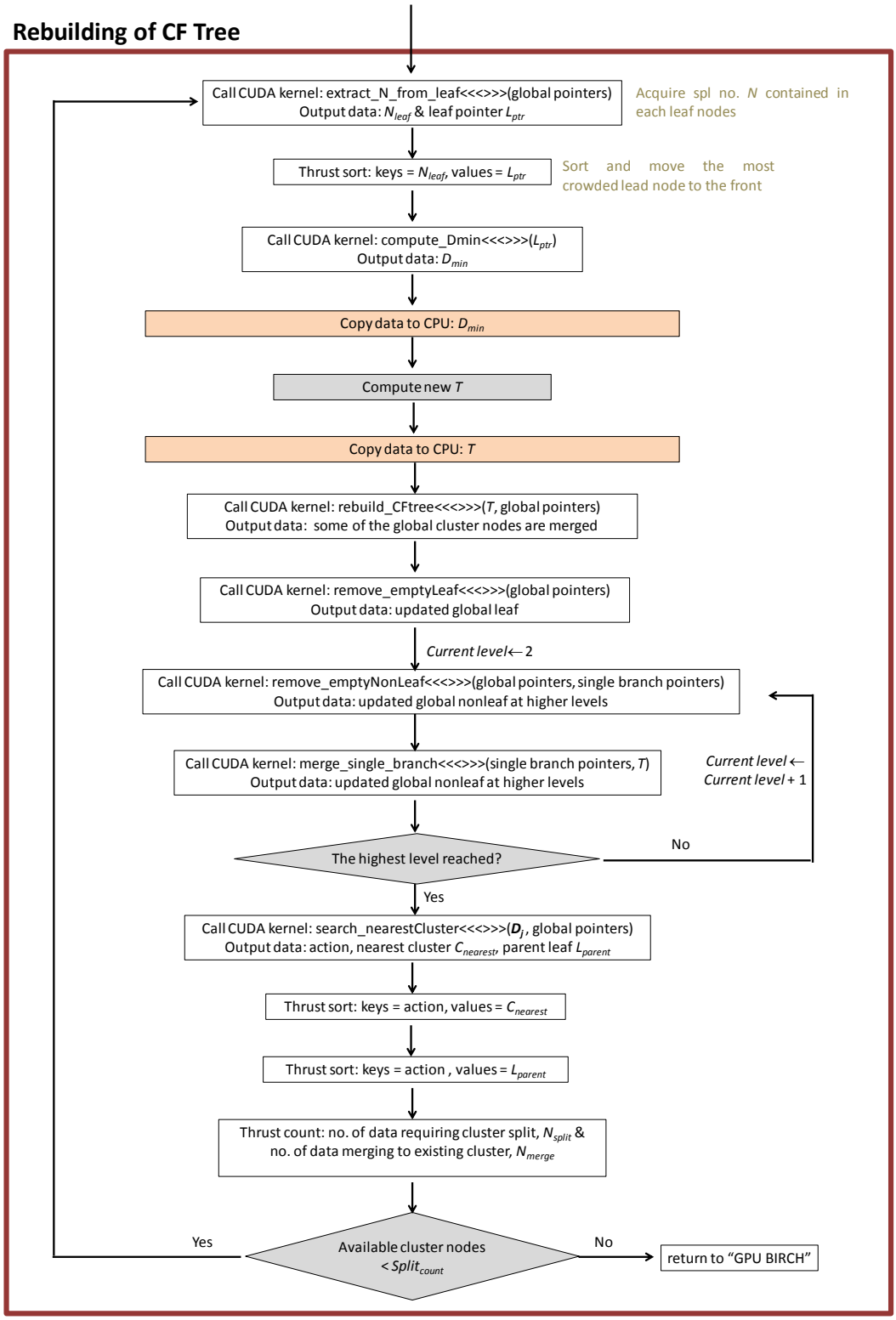

Figure 6-11: Procedure of sub-routine 'rebuilding of CF tree'.

6.3. Performance Evaluation

In this work, our main goal is to develop an efficient BIRCH algorithm for effective online monitoring of chemical process, using GPU parallelization technique. A series of numerical experiments, using both synthetic data and simulated process data, are carried out to evaluate the performance of the proposed hybrid CPU-GPU-BIRCH algorithm. In an online monitoring application, the main function of BIRCH is to provide accurate summary of large data, which in turn allows the construction of correct data models for different process states. The accuracy of data summarization can be measured based on a testing dataset which has not seen by BIRCH, by comparing the known and estimated process state. Considering the true labels and predicted labels as 2 partitions (referred to as *P*1 and *P*2), Jaccard coefficient (JC) can be used to measure the accuracy of clustering and also data summarization. JC is defined as follows,

$$JC = \frac{n_{11}}{n_{11} + n_{10} + n_{01}} \quad (6\text{-}1)$$

where, n_{11} refers to the number of pairs of objects which are in the same cluster in both *P*1 and *P*2, n_{10} denotes the number of pairs of objects assigned to the same cluster in *P*1 but not in *P*2, n_{01} is the number of pairs of objects assigned to the same cluster in *P*2 but not in *P*1. Note that the computation of JC is expensive, as it involves complexity of $O(N^2)$.

Moreover, in order to validate the effectiveness of the hybrid BIRCH algorithm in detecting slow process change, a gradual change is simulated in both synthetic and simulation data. To avoid biases towards high dimensional data, our performance validation is based on normalized dissimilarity value. As clusters are used as the data models $M_{i,s}$, normalization can be achieved by dividing the dissimilarity value *d* by data

dimensionality. The sensitivity of gradual change detection of the hybrid method will be established based on the sequential method. It should be noted that speedup realized on individual data window is of higher interest to the online use of BIRCH; whereas an overall speedup is more relevant for offline BIRCH application, which is used to generate clustering seeds.

Experimental results reported in this work are obtained based on a computing system containing 64-bit Intel Xeon CPU with processor speed of 3.20GHz, 12 GB RAM, and an Nvidia Quadro2000 GPU card. Taking advantage of the well-established functionality offered by MATLAB, the two algorithms, CPU-BIRCH and CPU-GPU-BIRCH, are wrapped into two separate MATLAB MEX functions. While auxiliary computations, including post-BIRCH clustering, JC computation, dissimilarity measurement, can also be executed on MATLAB platform. Direct application of MATLAB k-means function is especially straightforward. Furthermore, the function provides special feature of ignoring cluster seeds that are too far from the samples, by setting parameter 'emptyaction' as 'drop'. This feature prevents rigid clustering effect of using cluster seeds, i.e. it is not necessary to form *k* clusters if the data are collected from number of process states smaller than *k*.

The algorithm parameters employed in the experiments are specified as below. Euclidean distance is used in both approaches throughout all the experiments; the maximum total number of node *M*' is set at 15000; the size of *delay-split* variable for sequential approach is equal to the batch size of the GPU approach; the initial batch size for GPU-BIRCH is 64. A further point is that both synthetic data and simulation data is normalized to the range [0, 1] prior to analysis. As online data is normalized based on

the minimum and maximum values determined in the historical dataset, the normalized values could be found outside the range of [0,1].

6.3.1. Testing using synthetic data

Synthetic data offers the flexibility in covering a wide range of dimensionality, number of cluster, sample size, and deviation mode. Considering the use of Euclidean distance in BIRCH, normally distributed data is appropriate for evaluating the performance of the proposed approach. In (Zhang et al., 1996), the authors propose using the concept of grid for locating cluster centers. This strategy assures non-overlapping clusters, but its application for generating high dimensional data is rather complicated. In this work, the cluster centers are allocated along the diagonal path. A simple example is given in Table 6-1, where there are 3 clusters of data with dimensionality of 4. Besides the location of centers, variance is another parameter required for generating normally distributed data. The variance is chosen randomly from the range of [0.7, 1.5] for each cluster generation. Upon completion of data generation, the data is normalized across all clusters to the range of [0, 1]. In this work, we use 4 clusters for synthetically generated data.

Table 6-1: An example showing 4 cluster centers allocated along the diagonal plane.

# dimension	Cluster no./Center location		
	1	2	3
1	0	0.5	1
2	0	0.5	1
3	0	0.5	1
4	0	0.5	1

A number of synthetic datasets are generated to assume problems with different number of normal process states (or number of clusters) and dimensionality.

The total size of each dataset is set at 10 GB, and the number of patterns or samples is computed as follows,

$$no.of\ pattern = \frac{10 \times 10^9}{d \times data\ size} \quad (6\text{-}2)$$

where, *d* is the dimensionality and data size equal to 4 bytes (i.e. single precision data) is used throughout this work. Due to the expensive computational cost of JC, only 10% of the dataset (i.e. about 1 GB) is used as testing data. For small *d*, the number of samples can be extremely large, so a smaller fraction of 0.1% will be used for *d* below 256. The remaining 90% or 0.99% of data serves as the training data. At both online and offline stages, 0.5 GB is used as the window size.

The performance evaluation on synthetic data is carried out in 2 parts, which include studying the effect of parameters on computational performance in offline BIRCH applications, and investigating BIRCH performance in detecting gradual change during online monitoring.

6.3.1.1. **Effect of parameters**

First of all, we study the effect of several parameters on the performance of both sequential and hybrid BIRCH algorithms. Due to the complex operations involved in CUDA kernels, optimal thread configurations are determined through a series of numerical experiments. From the discussion on CUDA kernel design, we know that the number of blocks used in each CUDA kernel is determined dynamically by the number of occupied nodes, splitting samples, and etc. Therefore, block size is the only variable under evaluation in the experiments. The CUDA kernels are tested from the minimum block size i.e. 32 to the maximum allowable block size which can be 1024 or smaller. Although block size of 1024 is the device limitation for GPU with compute capability of

2.X, CUDA kernels which assign each thread with private space in shared memory may be limited to a smaller block size. It can be seen from Table 6-2, the overall computing time is significantly reduced as the block size of *search_nearestCluster* decreases from the maximum 1024 to 512, and it reaches the optimal performance at block size 256. Apart from that, block sizes of all other CUDA kernels do not have notable impact on the overall efficiency performance. Hence, the maximum block size is generally chosen for these CUDA kernels, as highlighted in yellow in Table 6-2. Note that the experimental results reported in Table 6-2 are based on dimensionality of 1792.

Table 6-2: Performance of BIRCH-related CUDA kernels at various thread configurations.

CUDA Kernel	Block Size	CPU-GPU-BIRCH Computing Time (s)
init_CFtree	64	114.140
	128	114.183
	256	114.077
	512	115.767
search_nearestCluster	64	92.241
	128	70.912
	256	67.562
	512	83.487
	1024	115.767
cluster_merging_splitting	64	119.700
	128	115.502
	256	114.694
	512	115.767
update_parent	64	118.622
	128	109.574
	256	108.755
	512	107.778
	1024	115.767
splitting	64	116.323
	128	116.085
	256	115.704
	512	115.767
compute_child2child_dist	64	114.227
	128	114.075
	256	114.116
	512	114.112
	1024	115.767
extract_N_from_leaf	128	114.038
	256	113.884
	512	113.957
	1024	115.767
compute_Dmin	64	114.166
	128	114.195
	256	114.242
	512	113.839
	1024	115.767
rebuild_CFtree	64	114.128
	128	113.950
	256	114.007
	512	113.978
	1024	115.767
remove_emptyLeaf & remove_emptyNonLeaf	64	114.283
	128	114.287
	256	114.089
	512	114.283
	1024	115.767
merge_single_branch	128	114.238
	256	114.169
	512	114.104
	1024	115.767

The maximum number of nodes *M*' allowable in CF-tree influences the performance of BIRCH in terms of the number of tree rebuilding, final threshold value, and also total computing time. Table 6-3 summarizes the performance of both CPU and hybrid BIRCH algorithms at varying *M*', in the range of [2500, 150000]. More stable performance is achieved by the hybrid BIRCH method, by having gradually increasing computing time, final number of clusters, and final threshold value, as *M*' increases. On the contrary, CPU BIRCH performance is rather unpredictable, especially for the total computing time and final number of clusters. Both methods show comparable final threshold values and achieve perfect clustering accuracy (i.e. JC=1). Note that the experimental results reported in Table 6-3 are based on dimensionality of 1792, and thread configurations determined from Table 6-2.

Table 6-3: Effect of maximum number of nodes on CPU and hybrid BIRCH algorithms.

	Max no. of nodes *M'*	Computing Time (s)	Speedups	Final no. of cluster nodes	Final *T* value
CPU-BIRCH	2500	410.492	11.62	6	0.884135
CPU-GPU-BIRCH		35.314		2337	0.867684
CPU-BIRCH	5000	615.796	14.46	448	0.849176
CPU-GPU-BIRCH		42.591		4532	0.867684
CPU-BIRCH	7500	490.936	8.58	142	0.847994
CPU-GPU-BIRCH		57.252		7496	0.867684
CPU-BIRCH	10000	623.119	10.10	1901	0.843245
CPU-GPU-BIRCH		61.700		9025	0.867684
CPU-BIRCH	15000	682.381	10.29	52	0.860510
CPU-GPU-BIRCH		66.302		9868	0.880077

The *delay-split* feature in CPU BIRCH algorithm has similar function to the batch processing feature in hybrid approach; that is to allow peeking into future samples. Hence, they are examined jointly for comparison purpose. The experimental results presented in Table 6-4 have once again validated the stable performance of GPU-based approach. As the batch size increases, steady increasing trends are observed for final number of clusters obtained by GPU approach. Larger batch size provides farther

view into the future samples, which promises for better arrangement of nodes in the CF-tree, thus offering finer clusters with smaller *T*. On the other hand, the effect of delay-split size is less predictable, in CPU approach. The final number of clusters obtained from CPU-BIRCH is much smaller than those obtained from the hybrid approach, and this could be a problem for further data analysis which may require certain volume of samples. Both methods show comparable final threshold values and achieve perfect clustering accuracy (i.e. JC=1), in all the experiments shown in Table 6-4. Note that the experimental results reported in Table 6-4 are based on dimensionality of 1792, optimal thread configurations obtained previously, and *M*'=15000.

Table 6-4: Effect of delay-split size (or batch size) on CPU-BIRCH (or hybrid BIRCH) algorithm.

	CPU: Delay-split size or GPU: batch size	Computing Time (s)	Final no. of cluster nodes	Final *T* value
CPU-BIRCH	0	719.999	4	0.844141
	200	679.962	1678	0.843245
	500	710.373	1502	0.843245
	1600	682.381	52	0.860510
	3200	705.249	44	0.860510
	4800	1130.000	181	0.860510
CPU-GPU-BIRCH	1600	66.302	9868	0.880077
	3200	61.490	13578	0.868020
	4800	53.169	13865	0.867684

In this work, the last parameter to be evaluated is data dimensionality. As has been mentioned early, the CUDA kernels are designed for addressing large dimensional problems, which is achieved by mapping CUDA threads to different variables (or elements) of a sample (or CF-vector). It can be seen from Table 6-5 that the computing time consumed by hybrid method generally increases as dimensionality decreases. This can be explained by the fact that the number of samples becomes larger at lower dimension, so as to make up for the total 10 GB data size, according to Equation (6-2). Although there is no obvious trend observed from the CPU approach, the computing

time is generally getting larger as dimensionality increases. It can be attributed to the repeated access of the same sample and tree nodes, for instance, the same sample is accessed during searching, merging or splitting, and parent node updating, and some nodes in the CF-tree may be accessed more than one time for searching and updating. As sequential method iterate through every dimension, increasing in dimensionality has dominant effect. Furthermore, the number of cluster nodes obtained from CPU-BIRCH tends to be lesser, particularly at dimension 256, 512, 768, and 1280. A possible reason for the unstable performance of CPU-BIRCH is that it is more sensitive to the sequence of data order. The exceptionally high threshold value at dimension 256 could be due to the same reason.

The proposed hybrid BIRCH algorithm outperforms the sequential counterpart by providing finer cluster nodes, enhanced computational efficiency with speedups range from 1.75 to 30.81, and comparable clustering accuracy where both methods achieve JC=1. Although at low dimension of 32, the GPU-based BIRCH is slower than the sequential method, satisfactory accuracy is still obtained. Note that the experimental results reported in Table 6-5 are based on batch size (or delay-split size) of 4800, *M*'=15000, and optimal thread configurations obtained previously; except that smaller block sizes containing 64 threads are used for low dimensions of 32 and 64.

Table 6-5: Performance of the CPU and hybrid BIRCH algorithms at various dimensionalities.

	Dimension	Computing Time (s)	Speedups	Final no. of cluster nodes	Final *T* value
CPU-BIRCH	32	460.475	0.81	1326	0.122814
CPU-GPU-BIRCH		570.074		1803	0.186470
CPU-BIRCH	64	509.679	1.75	15000	0.150357
CPU-GPU-BIRCH		291.846		14063	0.148701
CPU-BIRCH	128	808.533	2.21	11234	0.203730
CPU-GPU-BIRCH		366.128		14997	0.206173
CPU-BIRCH	256	528.525	4.06	4	4.503574
CPU-GPU-BIRCH		130.204		10056	0.311283
CPU-BIRCH	512	524.759	8.22	4	0.512881
CPU-GPU-BIRCH		63.839		7407	0.464331
CPU-BIRCH	768	606.394	14.09	4	0.530206
CPU-GPU-BIRCH		43.051		4816	0.589475
CPU-BIRCH	1024	733.692	15.46	4561	0.623849
CPU-GPU-BIRCH		47.465		11192	0.653548
CPU-BIRCH	1280	743.888	16.61	5	0.786242
CPU-GPU-BIRCH		44.791		11003	0.736693
CPU-BIRCH	1536	1200.000	30.81	3179	0.780714
CPU-GPU-BIRCH		38.952		5742	0.832203
CPU-BIRCH	1792	1130.000	21.25	181	0.860510
CPU-GPU-BIRCH		53.169		13865	0.867684
CPU-BIRCH	2048	1030.000	28.06	892	0.915374
CPU-GPU-BIRCH		36.703		5275	0.972107

6.3.1.2. **Online BIRCH**

The synthetic dataset which is used for assessing online monitoring performance of BIRCH is characterized by high dimensionality of 2048 and 4 process states (or clusters); 2.4GB data is generated for each cluster. Based on data window size of 0.5GB, the entire 9.6GB data is distributed into 18 data windows with some windows contain data belonging to the same cluster while others consists of data mixture, as summarized in Table 6-6. A simulated gradual change is effected on the first cluster whereby the initial half of the 2.4GB data is collected from normal operating condition, while the later half demonstrates a slow change in the first dimension. The shifted

process data is seen starting in the middle of second window till fifth window. A stepwise increment of 0.05% with 5000 samples size is used.

Table 6-6: True cluster membership of synthetic data in online data windows.

Data window#	Cluster membership of data
1	1
2	1
3	1
4	1
5	1,2
6	2
7	2
8	2
9	2,3
10	3
11	3
12	3
13	3
14	3,4
15	4
16	4
17	4
18	4

We examine the performance of BIRCH in detecting gradual change based on the dissimilarity measurements obtained from the 18 data windows, as plotted in Figure 6-12, for both the CPU and hybrid BIRCH approach. It can be seen from the region marked by 2 vertical green lines that both the sequential and hybrid BIRCH approaches successfully reflect the process change by showing increasing trends of dissimilarity values. However, significant dissimilarity values are also seen in data windows outside the gradual change range. For data window number 2 and 11, the high dissimilarity values detected by the CPU-BIRCH are caused by the abnormally small number of cluster nodes resulted from CF-tree, as given in Table 6-7; where there are only 2 and 3 cluster nodes obtained, respectively. In this case, the data is over-compressed and resulting in information lost. On the other hand, at data window number 9, both

approaches show a sudden jump in dissimilarity value at the start of cluster ID 3. Investigation of the raw data reveals that this particular window contains an extremely skewed data distribution by having 0.01% (or 7 out of 55339) data belonging to cluster ID 3, while the remaining pertaining to cluster ID 2. Given these 7 data points are found slightly far from the center of cluster ID 3 detected offline, the dissimilarity value computed from them is not representative of the real process behavior.

Apart from the one-off deficiency at data window ID 9, GPU-based BIRCH show better performance than the sequential as it achieves more stable and unnoticeable dissimilarity values at data windows corresponding to normal operation, provides finer cluster nodes, and also attains enormously enhanced computational speedups ranging from 6.17 to 31.58. The superior performance is attributed to the algorithmic design of GPU-based BIRCH, wherein it is allowed to peek into a batch of data at each time. By having a wider view of the incoming data, the algorithm is able to generate a tighter and well-organized CF-tree. Though the delay-split feature in CPU-based BIRCH provides a similar functionality, once the delay-split array is fully-occupied, there will be no more viewing into future data.

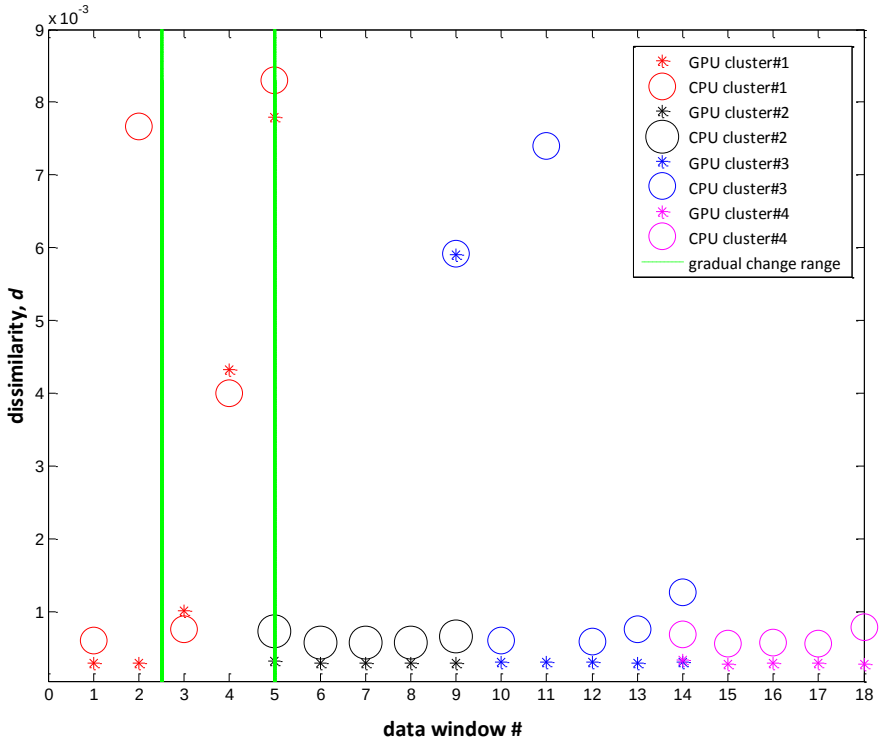

Figure 6-12: Dissimilarity values computed from the 18 synthetic data windows, for both CPU and hybrid BIRCH online application.

Table 6-7: Online performance of the CPU and hybrid BIRCH algorithms in 18 synthetic data windows.

	Data window#	Computing Time (s)	Speedups	Final no. of cluster nodes	Final *T* value
CPU-BIRCH	1	130.9834	21.36	254	0.905080
CPU-GPU-BIRCH		6.1327		14942	0.926791
CPU-BIRCH	2	83.9950	16.02	2	0.894441
CPU-GPU-BIRCH		5.2443		13140	0.930974
CPU-BIRCH	3	130.6619	22.79	2548	0.913371
CPU-GPU-BIRCH		5.7324		14874	0.924535
CPU-BIRCH	4	122.9683	31.58	2397	0.913506
CPU-GPU-BIRCH		3.8937		6029	0.955253
CPU-BIRCH	5	120.5004	6.23	964	0.908127
CPU-GPU-BIRCH		19.3488		7879	0.953495
CPU-BIRCH	6	129.9169	30.84	2140	0.913589
CPU-GPU-BIRCH		4.2122		8144	0.948954
CPU-BIRCH	7	118.9778	27.33	2626	0.914822
CPU-GPU-BIRCH		4.3532		9112	0.945061
CPU-BIRCH	8	122.7449	28.19	2342	0.910776
CPU-GPU-BIRCH		4.3548		8663	0.946673
CPU-BIRCH	9	118.7098	18.45	1826	0.909360
CPU-GPU-BIRCH		6.4336		14763	0.927780
CPU-BIRCH	10	134.8088	27.36	2273	0.912752
CPU-GPU-BIRCH		4.9279		10133	0.941493
CPU-BIRCH	11	124.0431	31.02	3	0.904626
CPU-GPU-BIRCH		3.9992		5863	0.958636
CPU-BIRCH	12	120.3278	23.77	2758	0.909641
CPU-GPU-BIRCH		5.0621		8593	0.946372
CPU-BIRCH	13	104.7925	19.54	1971	0.905996
CPU-GPU-BIRCH		5.3627		12805	0.934182
CPU-BIRCH	14	119.4594	6.17	1253	0.908239
CPU-GPU-BIRCH		19.3481		10253	0.945741
CPU-BIRCH	15	134.9836	23.92	2385	0.911285
CPU-GPU-BIRCH		5.6426		11777	0.933907
CPU-BIRCH	16	128.9014	27.90	2291	0.904564
CPU-GPU-BIRCH		4.6195		10905	0.936965
CPU-BIRCH	17	123.9921	17.98	2879	0.913103
CPU-GPU-BIRCH		6.8944		14268	0.929326
CPU-BIRCH	18	87.9097	15.70	965	0.905422
CPU-GPU-BIRCH		5.5976		13730	0.929604

6.3.2. **Application to simulated oil and gas production process**

A Kongsberg simulator of oil and gas offshore production process is employed in this work. Kongsberg has received credibility for distributing process simulators and real-time systems for oil and gas fields. An important reason for choosing the Kongsberg simulator is that it represents an adequately complex and realistic system. Essentially, oil, water and gas travel from the reservoir to the surface through the oil wells, under their own pressure or with the help of water or produced gas re-injection. At the surface, production from the well is routed to a separation system from which it is segregated into 3 basic components, i.e. oil, gas and water. As presented in Figure 6-13, the separation system comprises several key subsystems, which include oil separation, gas compression, gas dehydration, and gas reinjection sub-systems.

These key sub-systems are discussed briefly in the following. The well production is firstly sent to a conventional three-stage process, at a sequentially reduced pressure in each stage, namely High Pressure Separator, Medium Pressure Separator, and Low Pressure Separator. Produced water is separated at the H.P. separator while final dewatering of crude oil is carried out at a Coalescer. Crude generated from H.P. separator is further refined at M.P. and L.P. separators to meet certain specification, before it is collected at the oil storage tanks. The test separator is available to allow individual wells to be examined. Produced water which is released from H.P. separator is cleaned and degassed to required levels prior to discharging to a disposal well. In Kongsberg oil and gas simulator, a single gas processing train is employed in which gas collected from the oil separation system is compressed and dehydrated prior to injection or export into gas pipeline. The process model uses a two-stage compression system, where each stage consists of a cooler, a scrubber and a compressor. Based on the outlet pressure of the scrubber, oil output from the scrubber is sent back to either H.P. or L.P. separator. Gas at increased pressure is then sent to the

gas dehydration system whereby further water removal is achieved on a glycol contactor unit accompanying with cooler and scrubbers. The refined gas goes through another compressor prior to entering the gas export pipeline or gas re-injection system where further compression is required.

There is a total of 1701 process variables comprising of various measurements (e.g. temperature, pressure, flow, level), and opening positions for various valve types (e.g. choke valve, flow wing valve, bypass valve, gate valve, isolation valve). In this work, process data of all 1701 variables is collected at the rate of 3 milliseconds, using Microsoft Excel. In view of real-world gradual change detection applications, it would be much more practical to make use of existing process data instead of collecting a separate dataset. In general, fast data generation speed is required especially for the purpose of process control and monitoring, data rate of 3 milliseconds is thus assumed here.

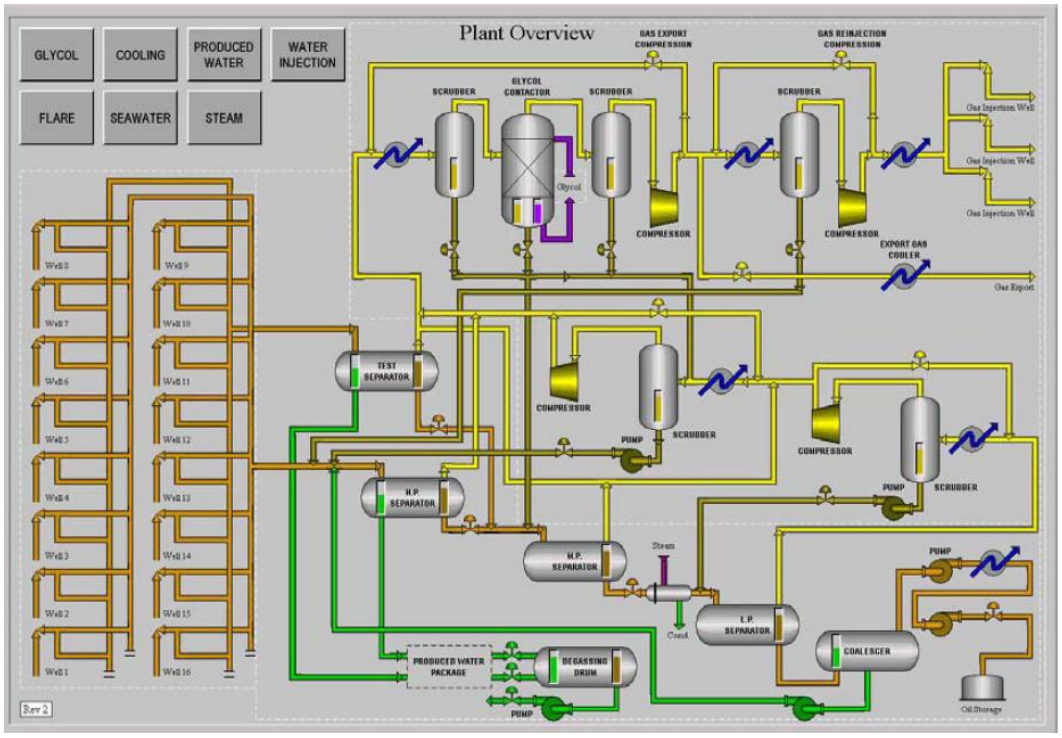

Figure 6-13: Plant overview of Kongsberg simulator for oil and gas production process[11].

A case study simulating 7 normal process states is employed, in which the process states are associated to different combinations of oil wells, as summarized in Figure 6-14. The overview of Kongsberg wells system is given in Figure 6-15. There are approximately 70,000 samples collected for each process state. As we are interested in extremely large data size, the sample size is doubled to 140,000 (or 1GB) by adding white noise at signal to noise ratio of 70. Once the normal operation data has been collected for the 7 process states, the opening of the choke valve positioned at the outlet of well 11 is intentionally reduced to simulate a fouling and clogging scenario in process state E. Note that choke valve is a kind of control valve which generally used in oil and gas production wells for controlling the flow of well fluids. The reduction of valve opening is carried at the rate of 5% for each 10,000 samples until it reaches 0%. As the original opening under normal operation is 50%, there are additional 100,000 samples

[11] Courtesy: (Kongsberg)

collected for cluster E. Again, the simulated data is quadruplicated through adding noise. As a result, there are a total of 400,000 samples (or 3.2GB) representing a slow change in the oil and gas production process. In GPU-based BIRCH, all the data is concatenated with 27 columns of zeros (i.e. padding) in order to increase dimensionality from 1701 to 1728, as the latter is divisible by 32 for optimal performance. All the experiments are performed using 4800 batch size (or delay-split size), 150000 M', and optimal thread configurations identified previously.

Process state ID	well # (0 means close; 1 means open)															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
C	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
D	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
F	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
G	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0

total number of opened wells = 6
total number of opened wells = 8

Figure 6-14: Well opening combinations associated with the 7 normal process states, using Kongsberg simulator.

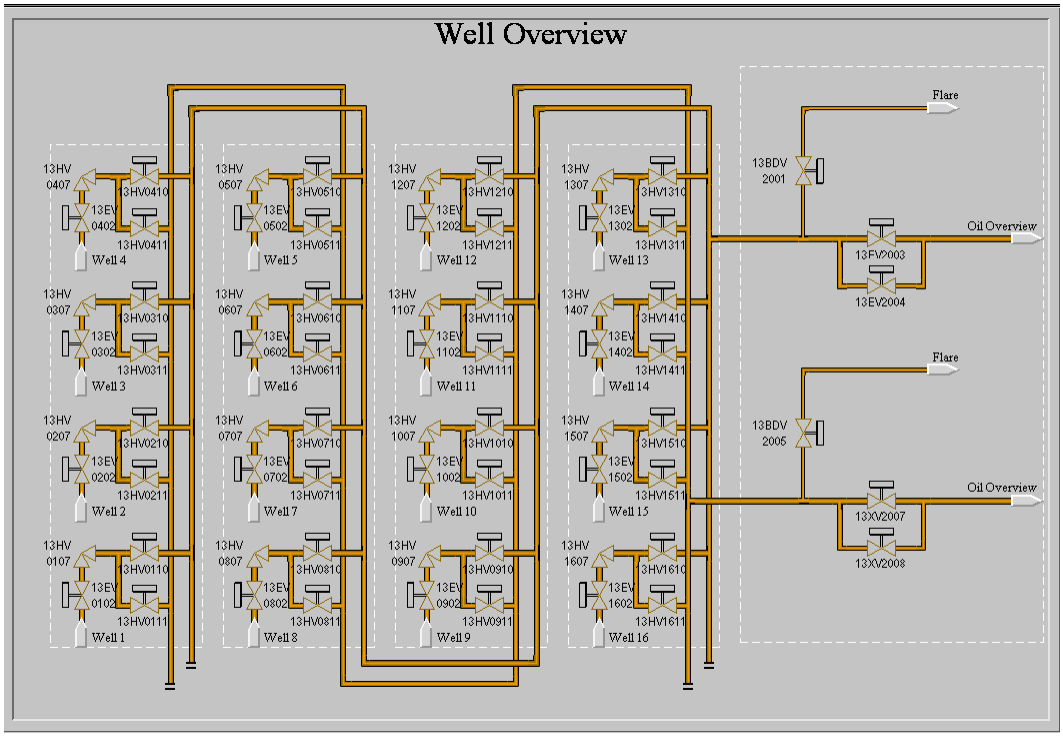

Figure 6-15: Overview of the well system in the Kongsberg simulator[12].

First of all, we look at the offline performance of BIRCH in generating cluster seeds and data models for the 7 normal process states. Given data window size of 0.5GB and 0.1% of total samples is separated out as testing samples, the 7 GB normal operating data is partitioned into 14 windows. The GPU-based BIRCH improves the computational efficiency of the sequential method by 10.06 times, as shown in Table 6-8. Furthermore, comparable clustering quality of JC=1 is also achieved by the hybrid approach.

The 7 cluster models obtained offline are then used for determining dissimilarity values during online monitoring. Note that the shifted data is divided into 5 windows with each consists of 0.64GB data. It can be seen from Figure 6-16 that both the sequential and hybrid methods successfully detect the gradual fouling effect of the

[12] Courtesy: (Kongsberg)

choke valve, by showing increasing dissimilarity values across the 5 data windows. Moreover, GPU-based method also demonstrates computational speedups ranging from 3.88 to 11.31. The achieved acceleration prevents data backlog and ensures timely feedback of anomaly. Although the threshold value obtained by GPU-BIRCH at the second window is slightly high, the amount of cluster nodes is still sufficiently large to serve as representative summary of the raw data.

Table 6-8: Offline performance of the CPU and hybrid BIRCH algorithms, based on Kongsberg simulation data.

	Computing Time (s)	Speedups	Final no. of cluster nodes	Final T value	JC
CPU-BIRCH	367.9032	10.06	5196	2.720166	0.9991
CPU-GPU-BIRCH	36.5546		6343	2.113261	1

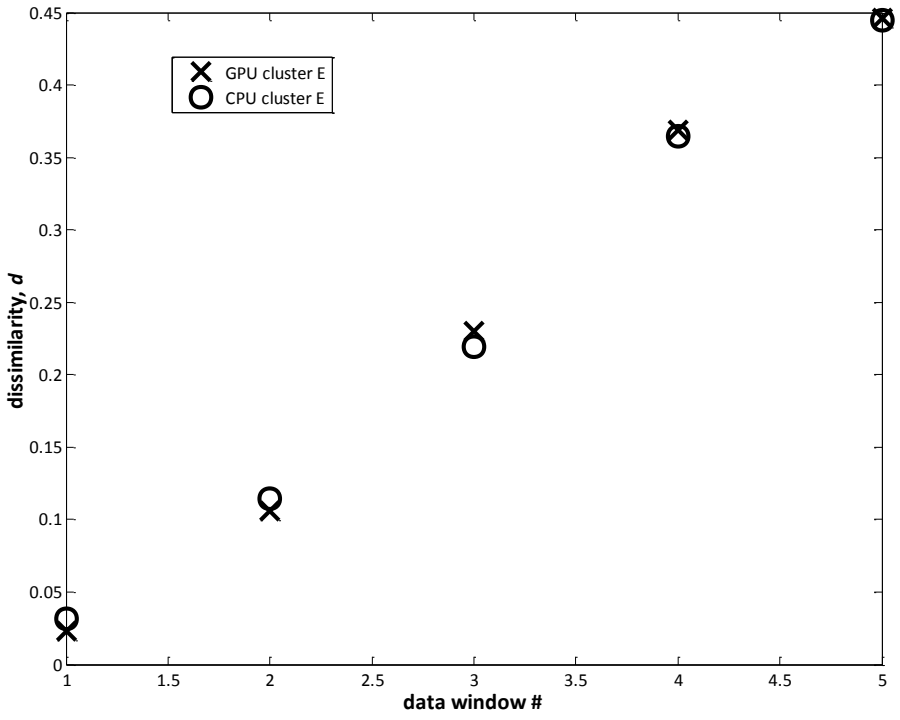

Figure 6-16: Dissimilarity values obtained from the CPU and GPU-based approaches, based on Kongsberg simulation data.

Table 6-9: Online performance of the CPU and hybrid BIRCH algorithms, based on Kongsberg simulation data.

	Data window#	Computing Time (s)	Speedups	Final no. of cluster nodes	Final T value
CPU-BIRCH	1	215.4363	11.31	14622	0.330398
CPU-GPU-BIRCH		19.0522		14496	0.588877
CPU-BIRCH	2	208.3054	4.36	11195	0.439866
CPU-GPU-BIRCH		47.7929		4256	3.983155
CPU-BIRCH	3	639.6968	3.88	12221	0.314146
CPU-GPU-BIRCH		165.0738		12093	0.791260
CPU-BIRCH	4	529.2386	6.86	9850	0.543664
CPU-GPU-BIRCH		77.1657		14615	0.541907
CPU-BIRCH	5	164.2366	7.24	14156	0.259741
CPU-GPU-BIRCH		22.6785		13447	0.525139

6.4. Summary

In modern chemical plants, effective online process monitoring requires handling of high dimensional data stream in an efficient manner. Although BIRCH is an effective clustering algorithm which is scalable to extremely large data, its computational speed is still unsuitably slow for online application. In this work, we propose a BIRCH-based online monitoring scheme, where the key function of BIRCH is to provide representative data summaries (in terms of micro clusters) for each data window. The summaries can then be further compressed into data models which can be used for detecting slow process change. We develop a GPU-based BIRCH algorithm, whereby the 2-level parallelization is essentially achieved by mapping CUDA blocks to samples or nodes, and mapping CUDA threads to dimensions. In order to overcome the limited parallelism of tree structure as well as to better utilize GPU resources, we incorporate several supporting features like global node pointers, memory pre-allocation, sorting, compacting, scanning, and batch processing of data.

The GPU-based BIRCH is evaluated using both synthetic and simulation data, where the original sequential method serves as the performance basis. Experimental

results show that GPU-based method provides more stable performance, in terms of the number of clusters and dissimilarity values (during online test). Note that stable dissimilarity values are crucial to prevent confusion or false alarm during process monitoring. Moreover, the hybrid BIRCH approach achieves satisfactory computational speedup at various parameter values e.g. total number of tree nodes, dimensionality, batch size; the maximum speedups achieved are 31.58. The only exception is seen at low dimension of 32. It has also been observed that the hybrid BIRCH method is exceptionally efficient when dealing well-separated clusters, as in the case of synthetic data. This could be explained by the fact that there is higher number of CF-tree rebuilding required when data is noisy and over-lapping.

Chapter 7. Conclusions and Future Work

7.1. Conclusions

Data mining is a computational tool used for extracting useful information from massive amounts of process data. The mined information which can be in the forms of models, patterns, or rules provide insights into the process, and enables accurate prediction about the process state, or detect anomaly. These offer opportunities for effective planning, process operations, process monitoring and control among others. However, conventional data mining algorithms which ignore the slow changes of real-world processes fail to sustain optimal performance. Real-time data mining seeks to overcome this obstacle through extracting timely, up-to-date information, continuously. As many data mining algorithms are computationally unsuitable for this task, real-time data mining applications remain challenging, particularly when the data involved is of high dimension, large size, and containing transients. GPU parallel computing technology offers enormous computing power at low cost and low power consumption and provides a promising and practical solution to real-time data mining problems. This thesis has sought to develop efficient GPU-based data mining algorithms for real-time applications. In particular NVIDIA's CUDA-enabled GPU was employed. We focused on three important chemical process applications, continuous optimization, real-time image analysis and data stream clustering.

First, we proposed a systematic procedure for developing efficient GPU-based Simulation-Optimization, Sim-Opt algorithms. A comprehensive structural analysis of Sim-Opt problems was presented, in which various types of data parallelism for different choice of techniques were revealed. Implementation of the proposed procedure was illustrated on a variable selection monitoring problem, where GA and

PCA were used as the optimization and simulation tool as proposed in the original work (Ghosh, 2014). For optimal efficiency, 5 CUDA kernels were constructed for executing the tasks of PCA modeling, objective evaluation, and non-dominated sorting. Major data parallelisms exploited in these CUDA kernels include concurrent handling of candidate solutions in GA, and simultaneous operations on training and testing samples. The excellent efficiency enhancement, up to 102 speedups, achieved by the proposed CPU-GPUGA-PCA algorithm was demonstrated through the benchmark Tennessee Eastman Challenge. A major limitation of the proposed framework is that GPU codes of many popular chemical process simulators such as Simulink, Aspen HYSYS, PROSIM, gPROMS are still not available both commercially and non-commercially. As the process units and components used in these simulators are usually a black box to the users, it is extremely difficult and tedious to construct the GPU counterpart. Therefore it might not be directly useful to the industry practitioners who rely heavily on those handy simulators on daily operations.

Second, a real-time multivariate image analysis solution was presented for particle size estimation, using GPU computing. Besides using the pixel-level parallelization which is a common strategy of standard GPU-based image processing techniques, our main contributions in this work included combining multiple feature extractors in single CUDA kernel to avoid unnecessary memory access, and exploring the use of GPU computing for multi-way PCA. Desirable real-time performance of the proposed GPU-based algorithm was demonstrated in a batch crystallization process, in which total computing time was reduced from 0.77 to 0.09 seconds per image; that is equivalent to 8.74 speedups. The proposed algorithm assumes that the images are stored in the host main memory and required to be moved to GPU memory for GPU processing. Since GPU is a graphics card and it should be able to directly read images

without going through the host memory, an improved version of GPU-MIA algorithm which is more computationally efficient can be obtained by incorporating such feature.

Third, a GPU-based online process monitoring scheme was proposed, wherein a scalable clustering algorithm called BIRCH was used to provide fine and accurate data summaries for effective model construction. In the proposed scheme, BIRCH algorithm was ported to GPU so that it can deal with high dimensional and fast-arriving online process data efficiently. The massive computing power of GPU was exploited primarily by processing data in batches, and treating tree nodes at the same level concurrently. Moreover, several supporting features like global node pointers, memory pre-allocation, sorting, compacting, and scanning of tree nodes, are proposed to overcome the limitation of tree structure in regards to data parallelism. Extensive testing of the proposed GPU-based approach using both synthetic and simulation data clearly showed its ability to provide stable, accurate and timely process monitoring performance. Furthermore, the proposed approach demonstrated much faster computational speed than the conventional CPU-based BIRCH approach, with speedups of nearly 32x.

A major constraint of BIRCH in continuous monitoring application is that it does not include the forgetting mechanism, thus it is sensitive to outlier. For that reason, in the proposed approach a new model is built for each data window. However, the selection of window size can be tricky. On the one hand, data window which is too small is also sensitive to outlier; on the other hand, data window that is too big tends to include outdated process information. Hence, an improvement of the proposed GPU-BIRCH approach can be achieved by replacing the BIRCH algorithm with its derivative clustering methods e.g. CluTree and CluStream. As it has been mentioned previously, BIRCH is used in this work because it serves as the fundamental basis to many other

data stream clustering methods, and it is relatively simpler to implement as an initial work in this field.

7.2. Future Work

In this section, some recommendations for future research extension are presented.

7.2.1. GPU performance on double precision computation

Experimental results reported in this thesis are entirely based on single precision (SP) computation for both CPU and GPU-based approaches. In chemical process operations, data collected from the plant e.g. sensor measurement, chromatogram, is usually noisy, thus SP computation is generally adequate. However, there are also cases where precise data analysis is crucial for ensuring safety or profitability. Process control applications in pharmaceutical plants is a classic example. As double precision (DP) calculations are fully supported by modern GPUs like NVIDIA's Quadro 2000, its implementation at first sight would appear to be straightforward. Existing GPU works which employ DP are mainly restricted to molecular and fluid dynamic simulation (Baghapour et al., 2014; Scott Le Granda et al., 2013; Zaspel & Griebel, 2013), in regards to chemical engineering applications. Since the data storage requirement of DP data and computing power consumed by DP operations are approximately double of their SP counterpart, it is necessary to perform thorough investigation and devise optimal GPU strategies for DP applications.

7.2.2. Improvement of input data quality

The synthetic data or simulation data used for performance evaluation in Chapter 4, Chapter 5, and Chapter 6 contain limited noise and few outliers, thus simple data normalization is adequate. Although data generated from Kongsberg simulator

resembles closely to real-world process data, the data collection is done in a way that only steady-state data is included. To thoroughly validate real-world applications of the proposed approaches, comprehensive data cleaning and pre-processing is needed for dealing with missing, noisy, and corrupted data. General data cleaning techniques have been widely reported in literature, including (Andritsos et al., 2006; Jeffery et al., 2006; Xiong et al., 2006; Žliobaite & Gabrys, 2014). Based on these studies, the development of data cleaning techniques, which are suitable for large-scale data mining of chemical process operations, can be explored in the future.

7.2.3. Integration with efficient data management system

In this thesis, efficiency performance evaluation is established based on the assumption that process data is always ready. However, from the point where data is generated at sensor or analytical instrument until it arrives at the computing system where data analysis is performed, it may take a time longer than the data analysis itself. Hence, efficient data management is another important issue that needs to be addressed in the future, in order to ensure timely capture of process information. Essentially, it requires two key components - a fast data network and an efficient database system. Considering the fast generation speed and large volume of plant data, traditional database systems which are designed for handling static data are likely to be unsuitable for the real-time data mining applications; more advanced database systems like the data stream management system discussed in (Tu et al., 2013) should be evaluated.

7.2.4. Multi-GPUs computing system

In this age of big data, the volume of data generated in modern chemical industries is growing exponentially due to advances in sensor technology, analysis

technique and expansion of analysis scope from plant-wide to regionally or even globally, the memory space in a standard GPU device can be easily exceeded. As long as the data mining algorithm is not scalable, once data size exceeds the main memory space of GPU, the data needs to be moved in and out of GPU. Memory transfer between the host and GPU is extremely expensive, and it can severely deteriorate the performance of GPU applications. In this case, multi-GPUs computing systems emerge as an alternative solution to preserve efficiency. On the other hand, although scalable algorithms such as BIRCH which is discussed in Chapter 6 are capable of handling unlimited amount of data theoretically, they may fail to meet the real-time requirement if data size is too large. Therefore, multi-GPUs computing system may eventually become essential for future data mining applications. High-throughput omics data processing is one of the potential applications as it involves enormously growing data size. For instance, the Next Generation Sequencing technique can analyze up to billions of base pair per run, and the upcoming Third Generation Sequencing technique is aiming at further improvement of throughput.

Bibliography

AccelerEyes. (n.d.). ArrayFire. Retrieved 6 17, 2014, from Graphics: http://www.accelereyes.com/arrayfire/c/page_gfx.htm

Adä, I. B. (2013). EVE: A framework for event detection. Evolving Systems, 4 (1), 61-70.

Adams, S. P. (2007). Finite difference time domain (FDTD) simulations using graphics processors. Department of Defense - Proceedings of the HPCMP Users Group Conference 2007; High Performance Computing Modernization Program: A Bridge to Future Defense, DoD HPCMP UGC, art. no. 4438007, (pp. 334-338).

Aggarwal, C. H. (2005). On high dimensional projected clustering of data streams. Data Mining and Knowledge Discovery, 10 (3), 251-273.

Agrawal, R. T. (2013). Support vector clustering-based direct coherency identification of generators in a multi-machine power system. IET Generation, Transmission and Distribution, 7 (12), 1357-1366.

Aji, A. S.-S. (2004). Partial least square modeling for the control of refining processes on mid-distillates by near infrared spectroscopy. Oil and Gas Science and Technology, 59 (3), 303-321.

Akman, U. O. (2008). Hierarchical clustering analysis for the distribution of origanum-oil components in dense CO2. Korean Journal of Chemical Engineering, 25 (2), 329-344.

Alon, U. B. (1999). Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. Proceedings of the National Academy of Sciences of the United States of America, 96 (12), 6745-6750.

Anderson, J. L. (2008). General purpose molecular dynamics simulations fully implemented on graphics processing units. Journal of Computational Physics, 227 (10), 5342-5359.

Andrecut, M. (2009). Parallel GPU implementation of iterative PCA algorithms. Journal of computational biology : a journal of computational molecular cell biology, 16 (11), 1593-1599.

Andritsos, P. F. (2006). Clean answers over dirty databases: A probabilistic approach. Proceedings - International Conference on Data Engineering, 2006, art. no. 1617398, (p. 30).

Arora, N. R. (2010). Fast sensitivity computations for trajectory optimization. Advances in the Astronautical Sciences, 135, (pp. 545-560).

Asano, S. M. (2009). Performance comparison of FPGA, GPU and CPU in image processing. FPL 09: 19th International Conference on Field Programmable Logic and Applications, art. no. 5272532, (pp. 126-131).

Assenhaimer, C. M. (2014). Use of a spectroscopic sensor to monitor droplet size distribution in emulsions using neural networks. Canadian Journal of Chemical Engineering, 92 (2), 318-323.

Azadivar, F. (1999). Simulation optimization methodologies. Winter Simulation Conference Proceedings, 1, (pp. 93-100).

B. Flannery, W. P. (1992). Numerical Recipies in C. Cambridge University Press, Cambridge UK, 2nd edition.

Babaei Pourkargar, D. A. (2010). Thermodynamic and transport properties estimation of a complicated mixture of hydrocarbons by fuzzy clustering methods. 19th International Congress of Chemical and Process Engineering, CHISA 2010 and 7th European Congress of Chemical Engineering, ECCE-7.

Baghapour, B. E. (2014). A discontinuous Galerkin method with block cyclic reduction solver for simulating compressible flows on GPUs. International Journal of Computer Mathematics, Article in Press. .

Bai, H. O. (2009). MAX-MIN ant system on GPU with CUDA. 2009 4th International Conference on Innovative Computing, Information and Control, ICICIC 2009, art. no. 5412455, (pp. 801-804).

Bai, H.-T. H.-L.-T.-S. (2009). K-means on commodity GPUs with CUDA. 2009 WRI World Congress on Computer Science and Information Engineering, CSIE 2009, 3, art. no. 5170921, (pp. 651-655).

Bakshi, B. (1998). Multiscale PCA with application to multivariate statistical process monitoring. AIChE Journal, 44 (7), 1596-1610.

Bezdek, J. H. (2004). Progressive sampling schemes for approximate clustering in very large data sets. IEEE International Conference on Fuzzy Systems, 1, (pp. 15-21).

Bhat, N. M. (1990). Use of neural nets for dynamic modeling and control of chemical process systems. Computers and Chemical Engineering, 573-582.

Biegler, L. C. (2002). Advances in simultaneous strategies for dynamic process optimization. Chemical Engineering Science, 575-593.

Biegler, L. Z. (2009). Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide dynamic optimization. Computers and Chemical Engineering, 575-582.

Blood, P. D. (2004). A versatile flow visualisation technique for quantifying mixing in a binary system: Application to continuous supercritical water hydrothermal synthesis (SWHS). Chemical Engineering Science, 59 (14), 2853-2861.

Böhm, C. N. (2009). Density-based clustering using graphics processors. International Conference on Information and Knowledge Management, Proceedings, (pp. 661-670).

Bolz, J. F. (2003). Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. ACM SIGGRAPH 2003 Papers, SIGGRAPH '03, (pp. 917-924).

Breschi, S. M. (2001). The geography of innovation and economic clustering: Some introductory notes. Industrial and Corporate Change, 10 (4), 817-833.

Brunet R., G.-G. G.-C. (2012). Hybrid simulation-optimization based approach for the optimal design of single-product biotechnological processes. Computers and Chemical Engineering, 37, 125-135.

Brunet R., R.-L. J.-G. (2012). Combined simulation-optimization methodology for the design of environmental conscious absorption systems. Computers and Chemical Engineering, 46. Article in Press., 205-216.

Bunin, G. (2014). On the equivalence between the modifier-adaptation and trust-region frameworks. Computers and Chemical Engineering, 71, 154-157.

Busciglio, A. V. (2009). Analysis of the bubbling behaviour of 2D gas solid fluidized beds. Part II. Comparison between experiments and numerical simulations via Digital Image Analysis Technique. Chemical Engineering Journal, 148 (1), 145-163.

Cano, A. Z. (2014). Speeding up multiple instance learning classification rules on GPUs. Knowledge and Information Systems, . Article in Press. .

Cao, F. E. (2006). Density-based clustering over an evolving data stream with noise. Proceedings of the Sixth SIAM International Conference on Data Mining, 2006, (pp. 328-339).

Cao, F. T. (2006). Scalable clustering using graphics processors. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4016 LNCS, (pp. 372-384).

Castaño-Díez, D. M. (2008). Performance evaluation of image processing algorithms on the GPU. Journal of Structural Biology, 164 (1), 153-160.

Catanzaro, B. S. (2008). Fast support vector machine training and classification on graphics processors. Proceedings of the 25th International Conference on Machine Learning, (pp. 104-111).

Chachuat, B. S. (2009). Adaptation strategies for real-time optimization. Computers and Chemical Engineering, 1557-1567.

Charu C. Aggarwal, J. H. (2003). A framework for clustering evolving data streams. VLDB '03: Proceedings of the 29th international conference on Very large data bases - Volume 29 , Volume 29. VLDB Endowment.

Chen, C. D. (2007). Bayesian clustering algorithms ascertaining spatial population structure: A new computer program and a comparison study. Molecular Ecology Notes, 7 (5), 747-756.

Chen, C. M. (2012). A GPU-accelerated approximate algorithm for incremental learning of Gaussian mixture model. Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012, art. no. 6270399, (pp. 1937-1943).

Chen, C. M. (2013). Towards a moderate-granularity incremental clustering algorithm for GPU. Proceedings - 2013 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2013, art. no. 6685679, (pp. 194-201).

Chen, Y. M. (2012). Simulation-optimization approach to clinical trial supply chain management with demand scenario forecast. Computers and Chemical Engineering, 40, 82-96.

Chiang, L. K. (2004). Fault diagnosis based on Fisher discriminant analysis and support vector machines. Computers and Chemical Engineering, 28 (8), 1389-1401.

Choi, S. P.-B. (2004). Process monitoring using a Gaussian mixture model via principal component analysis and discriminant analysis. Computers and Chemical Engineering, 1377-1387.

Cicciotti, M. X.-B. (2014). Simultaneous Nonlinear Reconciliation and Update of Parameters for Online Use of First-Principles Models: An Industrial Case-Study on Compressors. Computer Aided Chemical Engineering, 33, 457-462.

Cireşan, D. M. (2011). Flexible, high performance convolutional neural networks for image classification. IJCAI International Joint Conference on Artificial Intelligence, (pp. 1237-1242).

Clifton, C. (2014). "data mining." Encyclopaedia Britannica. Encyclopaedia Britannica Online. Retrieved 5 2, 2014, from Encyclopædia Britannica Inc.: http://www.britannica.com/EBchecked/topic/1056150/data-mining

Colin Cameron, A. G. (2011). Robust inference with multiway clustering. Journal of Business and Economic Statistics, 29 (2), pp. 238-249.

Corpet, F. (1988). Multiple sequence alignment with hierarchical clustering. Nucleic Acids Research, 16 (22), 10881-10890.

Cui, J.-Y. P. (2011). Fully 3D list-mode time-of-flight PET image reconstruction on GPUs using CUDA. Medical Physics, 38 (12), 6775-6786.

Cui, X. S. (2011). The GPU enhanced parallel computing for large scale data clustering. Proceedings - 2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2011, art. no. 6079384, (pp. 220-225).

Darakis, E. K. (2010). Microparticle characterization using digital holography. Chemical Engineering Science, 65 (2), 1037-1044.

David Kirk, W.-M. W. (2010). Programming Massively Parallel Processors. Springer Verlag.

De Calderon Anda, J. W. (2005). Multi-scale segmentation image analysis for the in-process monitoring of particle shape with batch crystallisers. Chemical Engineering Science, 60 (4), 1053-1065.

De P. Veronese, L. K. (2009). Swarm's flight: Accelerating the particles using C-CUDA. 2009 IEEE Congress on Evolutionary Computation, CEC 2009, art. no. 4983358, (pp. 3264-3270).

Deb, K. P. (2002). A fast elitist multi-objective gentic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation, 6 (2), 182-197.

Detroja, K. G. (2006). A possibilistic clustering approach to novel fault detection and isolation. Journal of Process Control, 1055-1073.

Dhillon, I. S. (2000). A parallel data-clustering algorithm for distributed memory multiprocessors. Large-Scale Parallel Data Mining, Lecture Notes in Artificial Intelligence, Volume 1759 (pp. 245–260). New York: Springer-Verlag.

Doan, X.-T. S. (2005). Augmented dynamic PCA approach for online monitoring of multi-phase batch processes. AIChE Annual Meeting, (pp. 6673-6695). Cincinnati.

Dong, J. W. (2013). Accelerating BIRCH for clustering large scale streaming data using CUDA dynamic parallelism. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8206 LNCS, (pp. 409-416).

Dunia, R. Q. (1996). Identification of Faulty Sensors Using Principal Component Analysis. AIChE Journal, 42 (10), 2797-2811.

Eden, M. J.-H. (2004). A novel framework for simultaneous separation process and product design. Chemical Engineering and Processing: Process Intensification, 43 (5), 595-608.

Edgar, R. (2010). Search and clustering orders of magnitude faster than BLAST. Bioinformatics, 26 (19), art. no. btq461, 2460-2461.

Eggers, J. K. (2008). Measurement of size and shape distributions of particles through image analysis. Chemical Engineering Science, 63 (22), 5513-5521.

Eisen, M. S. (1998). Cluster analysis and display of genome-wide expression patterns. Proceedings of the National Academy of Sciences of the United States of America, 95 (25), 14863-14868.

F. Farnstrom, J. L. (2000). True Scalability for Clustering Algorithms. SIGKDD Explorations.

Facco, P. B. (2009). Monitoring roughness and edge shape on semiconductors through multiresolution and multivariate image analysis. AIChE Journal, 55 (5), 1147-1160.

Fan, T. Z. (2013). A wireless electric field sensor based on a langasite resonator. Joint European Frequency and Time Forum and International Frequency Control Symposium (pp. 458-461). Prague: IEEE.

Fan, W. W. (2008). A framework for flexible clustering of multiple evolving data streams. International Journal of Advanced Intelligence Paradigms, 1 (2), 178-195.

Farber, R. (2011). CUDA Application Design and Development. Elsevier.

Farivar, R. R. (2008). A parallel implementation of K-means clustering on GPUs. Proceedings of the 2008 International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2008, (pp. 340-345).

Fayyad, U. P.-S. (1996). From data mining to knowledge discovery in databases. AI Magazine, 37-53.

Feng, Z. Z. (2007). A parallel hierarchical clustering algorithm for PCs cluster system. Neurocomputing, 70 (4-6), 809-818.

Fialka, O. Č. (2006). FFT and convolution performance in image filtering on GPU. Proceedings of the International Conference on Information Visualisation, art. no. 1648322, (pp. 609-614).

Fok, K.-L. W.-T.-L. (2007). Evolutionary computing on consumer graphics hardware. IEEE Intelligent Systems, 69-78.

Fujimoto, R. M. (1989). Parallel discrete event simulation. Winter Simulation Conference Proceedings, (pp. 19-28).

Fujimoto, R. M. (1990). Optimistic approaches to parallel discrete event simulation. Transactions of the Society for Computer Simulation, 7 (2), 153-191.

Funatsu, N. K. (2010). Fast parallel processing using GPU in computing L1-PCA bases. IEEE Region 10 Annual International Conference, Proceedings/TENCON, art. no. 5686614, (pp. 2087-2090).

Fung, J. M. (2008). Using graphics devices in reverse: GPU-based Image Processing and Computer Vision. 2008 IEEE International Conference on Multimedia and Expo, ICME 2008 - Proceedings, art. no. 4607358, (pp. 9-12).

Gaber, M. Z. (2005). Mining data streams: A review. SIGMOD Record, (pp. 18 - 26).

Gama, J. (2012). A survey on learning from data streams: current and future trends. Progress in Artificial Intelligence, 45-55.

Garcia, V. D. (2008). Fast k nearest neighbor search using GPU. 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops, art. no. 4563100.

Garcia, V. D. (2010). K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. Proceedings - International Conference on Image Processing, ICIP, art. no. 5654017, (pp. 3757-3760).

Garg, A. M. (2006). PBIRCH: A scalable parallel clustering algorithm for incremental data. Proceedings of the International Database Engineering and Applications Symposium, IDEAS, art. no. 4041640, (pp. 315-316).

Ge, Z. Y. (2009). Improved kernel PCA-based monitoring approach for nonlinear processes. Chemical Engineering Science, 64 (9), 2245-2255.

Geng, Z. Z. (2005). Multiscale Nonlinear Principal Component Analysis (NLPCA) and its application for chemical process monitoring. Industrial and Engineering Chemistry Research, 44 (10), 3585-3593.

Ghosh, K. R. (2014). Optimal variable selection for effective statistical process monitoring. Computers and Chemical Engineering, 60, 260-276 .

Goil, S. C. (1997). High performance OLAP and data mining on parallel computers. Data Mining and Knowledge Discovery, 391-417.

Goldberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning. Madison Wesley.

Golshan, M. B. (2005). A new approach to real time optimization of the Tennessee Eastman challenge problem. Chemical Engineering Journal, 33-44.

Gonzaga, J. M. (2009). ANN-based soft-sensor for real-time process monitoring and control of an industrial polymerization process. Computers and Chemical Engineering, 43-49.

Grabmeier, J. R. (2002). Techniques of cluster algorithms in data mining. Data Mining and Knowledge Discovery, 6 (4), 303-360.

Grahn, H. L. (2011). CudaRF: A CUDA-based implementation of random forests. Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA, art. no. 6126612, (pp. 95-101).

Gu, X. P. (2010). Implementation and evaluation of various demons deformable image registration algorithms on a GPU. Physics in Medicine and Biology, 55 (1), 207-219.

Gupta, A. M. (2003). Managing demand uncertainty in supply chain planning. Computers and Chemical Engineering, 1219-1227.

Halim, I. S. (2011). A knowledge-based simulation-optimization framework and system for sustainable process operations. Computers and Chemical Engineering, 35 (1), 92-105.

Han, E.-H. K. (2000). Scalable parallel data mining for association rules. IEEE Transactions on Knowledge and Data Engineering, 337-352.

Haque, I. P. (2010). SIML: A fast SIMD algorithm for calculating LINGO chemical similarities on GPUs and CPUs. Journal of Chemical Information and Modeling, 50 (4), 560-564.

He, Q. Q. (2005). A new fault diagnosis method using fault directions in Fisher discriminant analysis. AIChE Journal, 51 (2), 555-571.

Heng, Y. G. (2005). GPU-based volume rendering for medical image visualization. Annual International Conference of the IEEE Engineering in Medicine and Biology - Proceedings, 7 VOLS, art. no. 1615635, (pp. 5145-5148).

Henson, M. (1998). Nonlinear model predictive control: Current status and future directions. Computers and Chemical Engineering, 187-202.

Hore, P. H. (2007). Single pass fuzzy c means. IEEE International Conference on Fuzzy Systems, art. no. 4295372.

Hwu, W.-m. W. (2012). GPU Computing Gems, Volume 2. Elsevier.

Jason Sanders, E. K. (2010). CUDA by Example: An Introduction to General-Purpose GPU Programming. Ann Arbor: Edwards Brothers.

Jeffery, S. A. (2006). A pipelined framework for online cleaning of sensor data streams. Proceedings - International Conference on Data Engineering, 2006, art. no. 1617508, (p. 140).

Jošth, R. A.-K. (2012). Real-time PCA calculation for spectral imaging (using SIMD and GP-GPU). Journal of Real-Time Image Processing, 7 (2), 95-103.

Juang, C.-F. C.-C.-Y. (2011). Speedup of implementing fuzzy neural networks with high-dimensional inputs through parallel processing on graphic processing units. IEEE Transactions on Fuzzy Systems, 19 (4), art. no. 5744114, 717-728.

Kantzas, A. K. (1996). Monitoring the fluidization characteristics of polyolefin resins using x-ray Computer Assisted Tomography scanning. Chemical Engineering Science, 51 (10), 1979-1990.

Karuppiah, R. G. (2006). Global optimization for the synthesis of integrated water systems in chemical processes. Computers and Chemical Engineering, 650-673.

Kaur, S. K. (2012). Scalable clustering using PACT programming model. Proceedings - 12th IEEE International Conference on Data Mining Workshops, ICDMW 2012, art. no. 6406471, (pp. 424-430).

Kempkes, M. V. (2010). Measurement of 3D particle size distributions by stereoscopic imaging. Chemical Engineering Science, 65 (4), 1362-1373.

Khalil, A. P.-M.-P. (2010). Study of droplet size distribution during an emulsification process using in situ video probe coupled with an automatic image analysis. Chemical Engineering Journal, 165 (3), 946-957.

Khamene, A. B. (2006). Automatic registration of portal images and volumetric CT for patient positioning in radiation therapy. Medical Image Analysis, 10 (1), 96-112.

Kongsberg. (n.d.). Kongsberg Martime. Retrieved 6 3, 2014, from http://www.km.kongsberg.com/ks/web/nokbg0240.nsf/AllWeb/B2F29B3742D75297C1257315003C3F6F?OpenDocument

Kostopoulos, S. G. (2014). A pattern recognition system for prostate mass spectra discrimination based on the CUDA parallel programming model. Journal of Physics: Conference Series, 490 (1), art. no. 012144.

Kranen, P. A. (2008). Self-adaptive anytime stream clustering. Proceedings - IEEE International Conference on Data Mining, ICDM, art. no. 5360250, (pp. 249-258).

Kresta, J. V. (1991). Multivariate statistical monitoring of process operating performance. Canadian Journal of Chemical Engineering, 69 (1), 35-47.

Krüger, J. W. (2003). Linear algebra operators for GPU implementation of numerical algorithms. ACM Transactions on Graphics, 22 (3), (pp. 908-916).

Ku, W. S. (1995). Disturbance detection and isolation by dynamic principal component analysis. Chemometrics and Intelligent Laboratory Systems (pp. 179-196). Elsevier.

Laguna-Sánchez, G. O.-C.-C.-F.-C. (2009). Comparative study of parallel variants for a Particle Swarm Optimization algorithm implemented on a multithreading GPU. Journal of Applied Research and Technology, 7 (3), 292-309.

Langdon, W. H. (2008). GP on SPMD parallel graphics hardware for mega bioinformatics data mining. Soft Computing, 12 (12), pp. 1169-1183.

Larsen, P. M. (2009). The potential of current high-resolution imaging-based particle size distribution measurements for crystallization monitoring. AIChE Journal, 55 (4), 896-905.

Larsen, P. R. (2006). An algorithm for analyzing noisy, in situ images of high-aspect-ratio crystals to monitor particle size distribution. Chemical Engineering Science, 61 (16), 5236-5248.

Larsen, P. R. (2007). Model-based object recognition to measure crystal size and shape distributions from in situ video images. Chemical Engineering Science, 62 (5), 1430-1441.

Lee, J. M. (2003). On-line batch process monitoring using a consecutively updated multiway principal component analysis model. Computers and Chemical Engineering, 1903-1912.

Lee, J. Y. (2004). Nonlinear process monitoring using kernel principal component analysis. Chemical Engineering Science, 223-234.

Leischner, N. O. (2010). GPU sample sort. Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS , (p. art. no. 5470444).

Li, H. Y. (2012). Parallel based on cloud computing to achieve large data sets clustering. Proceedings - 2012 International Conference on Computer Science and Electronics Engineering, ICCSEE 2012, 1, art. no. 6187874, (pp. 411-415).

Li, J.-M. W.-J.-S.-X. (2007). An efficient fine-grained parallel genetic algorithm based on GPU-accelerated. Proceedings - 2007 IFIP International Conference on Network and Parallel Computing Workshops, NPC 2007, art. no. 4351594, (pp. 855-862).

Li, L. W. (1999). Three-dimensional image analysis of mixing in stirred vessels. AIChE Journal, 45 (9), 1855-1865.

Li, L. X. (2011). Research on clustering algorithm and its parallelization strategy. Proceedings - 2011 International Conference on Computational and Information Sciences, ICCIS 2011, art. no. 6086201, (pp. 325-328).

Li, S. W. (2009). Study on the data preprocessing of the questionnaire based on the combined classification data mining model. International Conference on E-learning, E-Business, Enterprise Information Systems, and E-Government, (pp. 217-220).

Li, W. G. (2006). Cd-hit: A fast program for clustering and comparing large sets of protein or nucleotide sequences. Bioinformatics, 22 (13), 1658-1659.

Li, Y. Z. (2010). Speeding up k-Means algorithm by GPUs. Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010, art. no. 5578441, (pp. 115-122).

Liang, S. L. (2009). A CUDA-based parallel implementation of K-nearest neighbor algorithm. CyberC 2009 - International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, 291-296.

Liao, Q. W. (2009). GPU accelerated Support Vector machines for mining high-throughput screening data. Journal of Chemical Information and Modeling, 2718-2725.

Lim, C. G. (2007). Bubble distribution and behaviour in bubbling fluidised beds. Chemical Engineering Science, 62 (1-2), 56-69.

Liu, F. H.-C.-H.-H. (2009). Efficient depth peeling via bucket sort. Proceedings of the HPG 2009: Conference on High-Performance Graphics 2009, (pp. 51-58).

Liu, G.-Q. L.-Q.-L. (2008). Experimental studies of particle flow dynamics in a two-dimensional spouted bed. Chemical Engineering Science, 63 (4), 1131-1141.

Liu, P. A. (2011). Accelerating chemical database searching using graphics processing units. Journal of Chemical Information and Modeling, 51 (8), 1807-1816.

Liu, W. S.-W. (2007). Streaming algorithms for biological sequence alignment on GPUs. IEEE Transactions on Parallel and Distributed Systems, 18 (9), 1270-1281.

Liu, Y. P.-K. (2004). Performance evaluation and characterization of scalable data mining algorithms. IASTED International Conference on Parallel and Distributed Computing and Systems, (pp. 620-625).

Liu, Y.-Q. L.-H.-H. (2006). Real-time 3D fluid simulation on GPU with complex obstacles. Ruan Jian Xue Bao/Journal of Software, 17 (3), 568-576.

Loop, C. B. (2006). Real-time GPU rendering of piecewise algebraic surfaces. ACM SIGGRAPH 2006 Papers, SIGGRAPH '06, (pp. 664-670).

M. P. Ekstrom, E. (1984). Digital Image Processing Techniques. Orlando, FL.: Academic Press.

Ma, C. W.-Q. (2011). GPU accelerated chemical similarity calculation for compound library comparison. Journal of Chemical Information and Modeling, 51 (7), 1521-1527.

Ma, W. A. (2009). A translation system for enabling data mining applications on GPUs. Proceedings of the International Conference on Supercomputing, art. no. 1542331, (pp. 400-409).

MacGregor, J. F. (1994). Process monitoring and diagnosis by multiblock PLS methods. AIChE Journal, 40 (5), 826-838.

Machlica, L. V. (2011). Fast estimation of gaussian mixture model parameters on GPU using CUDA. Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings, art. no. 6118944, (pp. 167-172).

Mahmood, A. S. (2013). Data mining techniques for wireless sensor networks: A survey. International Journal of Distributed Sensor Networks, 2013, art. no. 406316.

Malmberg, A. M. (2002). The elusive concept of localization economies: Towards a knowledge-based theory of spatial clustering. Environment and Planning A, 34 (3), 429-449.

Manssen, M. W. (2012). Random number generators for massively parallel simulations on GPU. European Physical Journal: Special Topics, 210 (1), 53-71.

Mariscal, G. M. (2010). A survey of data mining and knowledge discovery process models and methodologies. Knowledge Engineering Review, 137-166.

Martínez-Zarzuela, M. D.-P.-R.-H.-O.-G.-G. (2011). Multi-scale neural texture classification using the GPU as a stream processing engine. Machine Vision and Applications, 22 (6), 947-966.

Melab, N. C.-G. (2006). Grid computing for parallel bioinspired algorithms. Journal of Parallel and Distributed Computing, 1052-1061.

Mele, F. G. (2006). A simulation-based optimization framework for parameter optimization of supply-chain networks. Industrial and Engineering Chemistry Research, 45 (9), 3133-3148.

Méndez, C. C. (2003). Dynamic scheduling in multiproduct batch plants. Computers and Chemical Engineering, 27, (pp. 8-9).

Méndez, C. C. (2006). State-of-the-art review of optimization methods for short-term scheduling of batch processes. Computers and Chemical Engineering, 913-946.

Mhamdi, F. E. (2008). A new survey on knowledge discovery and data mining . Proceedings of the 2nd International Conference on Research Challenges in Information Science, RCIS 2008, art. no. 4632134, 427-432.

Misra, J. (1986). Distributed discrete-event simulation. ACM Computing Surveys (CSUR) Volume 18 Issue 1, 39-65.

Monnier, O. F. (1997). Model identification of batch cooling crystallizations through calorimetry and image analysis. Chemical Engineering Science, 52 (7), 1125-1139.

Moreland, K. A. (2003). GPU, The FFT on a., (pp. 112-119).

Mukhopadhyay, A. M. (2014). Survey of multiobjective evolutionary algorithms for data mining: Part II. IEEE Transactions on Evolutionary Computation, 18 (1), art. no. 6658840, 25-35.

Mussi, L. D. (2011). Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture. Information Sciences, 181 (20), 4642-4657.

Nahas, E. H. (1992). Nonlinear internal model control strategy for neural network models. Computers and Chemical Engineering, 1039-1057.

Ng, W. L. (2013). Sustainable supply network design through optimisation with clustering technique integration. Chemical Engineering Transactions, 35, 661-666.

Ng, Y. S. (2008). Multivariate temporal data analysis using self-organizing Maps. 1. Training methodology for effective visualization of multistate operations. Industrial and Engineering Chemistry Research, 47 (20), 7744-7757.

Ng, Y. S. (2008). Multivariate temporal data analysis using self-organizing Maps. 2. Monitoring and diagnosis of multistate operations. Industrial and Engineering Chemistry Research, 47 (20), 7758-7771.

Nikolopoulou, A. I. (2012). Hybrid simulation based optimization approach for supply chain management. Computers and Chemical Engineering, Article in Press.

Nomikos, P. M. (1994). Monitoring batch processes using multiway principal component analysis. AIChE Journal, 1361-1373.

Ntoutsi, I. Z.-P. (2012). Density-based projected clustering over high dimensional data streams. Proceedings of the 12th SIAM International Conference on Data Mining, SDM 2012, (pp. 987-998).

NVIDIA. (2012). NVIDIA CUDA C Programming Guide Version 4.2. NVIDIA.

NVIDIA. (n.d.). CUDA Toolkit 4.2 - archive. Retrieved 6 10, 2014, from https://developer.nvidia.com/cuda-toolkit-42-archive

NVIDIA. (n.d.). CUDA TOOLKIT DOCUMENTATION. Retrieved 6 12, 2014, from CUDA Toolkit v6.0 Profiler: http://docs.nvidia.com/cuda/profiler-users-guide/#axzz34OMu5tj0

NVIDIA. (n.d.). GPU-ACCELERATED APPLICATIONS. Retrieved 6 7, 2014, from http://www.nvidia.com/content/tesla/pdf/gpu-accelerated-applications-for-hpc.pdf

NVIDIA. (n.d.). NVIDIA CUDA ZONE. Retrieved 6 17, 2014, from NVIDIA Performance Primitives: https://developer.nvidia.com/NPP

NVIDIA. (n.d.). NVIDIA Nsight Visual Studio Edition. Retrieved 6 12, 2014, from https://developer.nvidia.com/nvidia-nsight-visual-studio-edition

NVIDIA. (n.d.). NVIDIA's Fermi: The First Complete GPU Computing Architecture. Retrieved 6 6, 2014, from http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf

Oullion, M. P. (2007). Industrial batch crystallization of a plate-like organic product. In situ monitoring and 2D-CSD modelling: Part 1: Experimental study. Chemical Engineering Science, 62 (3), 820-832.

Owens, J. L. (2007). A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 26 (1), 80-113.

P. Bradley, U. F. (1998). Scaling clustering algorithms to large databases. Fourth International Conference on Knowledge Discovery and Data Mining. AAAI Press.

Pahija, E. M. (2014). Assessment of control techniques for the dynamic optimization of (semi-)batch reactors. Computers and Chemical Engineering, 66, 269-275.

Pallarès, D. J. (2006). A novel technique for particle tracking in cold 2-dimensional fluidized beds - Simulating fuel dispersion. Chemical Engineering Science, 61 (8), 2710-2720.

Pan, L. G. (2008). Implementation of medical image segmentation in CUDA. 5th Int. Conference on Information Technology and Applications in Biomedicine, ITAB 2008 in conjunction with 2nd Int. Symposium and Summer School on Biomedical and Health Engineering, IS3BHE 2008, art. no. 4570542, (pp. 82-85).

Papenhausen, E. W. (2013). GPU-accelerated incremental correlation clustering of large data with visual feedback. Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013, art. no. 6691716, (pp. 63-70).

Papoulias, S. G. (1983). A structural optimization approach in process synthesis-II. Heat recovery networks. Computers and Chemical Engineering, 707-721.

Park, I. S. (2011). Design and performance evaluation of image processing algorithms on GPUs. IEEE Transactions on Parallel and Distributed Systems, 23 (1), art. no. 5477417, 91-104.

Patwary, M. P.-K. (2012). A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. International Conference for High Performance Computing, Networking, Storage and Analysis, SC, art. no. 6468492.

Peters, N. G. (2007). Real-time dynamic optimization of batch systems. Journal of Process Control, 261-271.

Pham, D. A. (2007). Clustering techniques and their applications in engineering. Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science, 221 (11), 1445-1459.

Pilkington, N. Z. (2010). An implementation of decision tree-based context clustering on graphics processing units. Proceedings of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH 2010, (pp. 833-836).

Pitas, I. (1993). Digital Image Processing Algorithms. New York: Prentice Hall Inc.

Polli, M. S. (2002). Bubble size distribution in the sparger region of bubble columns. Chemical Engineering Science, 57 (1), 197-205.

Pospichal, P. J. (2010). Parallel genetic algorithm on the CUDA architecture. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6024 LNCS (PART 1), (pp. 442-451).

Prabhu, R. (2008). SOMGPU: An unsupervised pattern classifier on Graphical Processing Unit. 2008 IEEE Congress on Evolutionary Computation, CEC 2008, art. no. 4630920, (pp. 1011-1018).

Prata, D. S. (2009). Nonlinear dynamic data reconciliation and parameter estimation through particle swarm optimization: Application for an industrial polypropylene reactor. Chemical Engineering Science, 3953-3967.

Preis, T. V. (2009). GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. Journal of Computational Physics, 228 (12), 4468-4477.

Psichogios, D. U. (1992). A hybrid neural network-first principles approach to process modelling. AICHE JOURNAL, 1499-1511.

Pyun, H. K. (2011). Monitor and diagnosis of LNG plant fractionation process using k-mean clustering and principal component analysis. Computer Aided Chemical Engineering, 29, pp. 1899-1903.

Qin, S. L. (2001). Detection and identification of faulty sensors in dynamic processes. AIChE Journal, 47 (7), 1581-1593.

Quaglia, A. G. (2015). Systematic network synthesis and design: Problem formulation, superstructure generation, data management and solution. Computers and Chemical Engineering, 72, , 68-86.

Radeke, C. G. (2010). Large-scale powder mixer simulations using massively parallel GPUarchitectures. Chemical Engineering Science, 65 (24), 6435-6442.

Redondo, J. G. (2011). Parallel evolutionary algorithms based on shared memory programming approaches. Journal of Supercomputing, 58 (2), 270–279.

Ren, J. S. (2005). Classified fuzzy association rules mining model and its application in aromatic hydrocarbon extraction. Huagong Xuebao/Journal of Chemical Industry and Engineering (China), 2137-2141.

Reyes-Labarta, J. C. (2012). A Novel Hybrid Simulation-Optimization Approach for the Optimal Design of Multicomponent Distillation Columns. Computer Aided Chemical Engineering, 30, 1257-1261.

Robilliard, D. M. (2009). High performance genetic programming on GPU. Proceedings of the 2009 Workshop on Bio-inspired Algorithms for Distributed Systems, BADS '09, (pp. 85-93).

Robilliard, D. M.-P. (2009). Genetic programming on graphics processing units. Genetic Programming and Evolvable Machines, 10 (4), 447-471.

Rodrigues, P. G. (2008). Hierarchical clustering of time-series data streams. IEEE Transactions on Knowledge and Data Engineering, 20 (5), art. no. 4407702, 615-627.

Rong, G. G. (2008). A classification framework for process operation optimization and its application in a triazophos plant. Chemical and Biochemical Engineering Quarterly, 22 (2), 233-243.

Ruiz, A. U. (2009). Non-rigid registration for large sets of microscopic images on graphics processors. Journal of Signal Processing Systems, 55 (1-3), 229-250.

Ryoo, S. R.-M. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, (pp. 73-82).

Sajdak, M. M. (2013). Biomass, biochar and hard coal: Data mining application to elemental composition and high heating values prediction. Journal of Analytical and Applied Pyrolysis, 104, 153-160.

Samant, S. X.-Ö. (2008). High performance computing for deformable image registration: Towards a new paradigm in adaptive radiotherapy. Medical Physics, 35 (8), 3546-355.

Santomaso, A. O. (2004). Mechanisms of mixing of granular materials in drum mixers under rolling regime. Chemical Engineering Science, 59 (16), 3269-3280.

Saraiva, P. M. (1992). Continuous process improvement through inductive and analogical learning. AIChE Journal, 161-183.

Sarkar, D. D.-T. (2009). In situ particle size estimation for crystallization processes by multivariate image analysis. Chemical Engineering Science, 64 (1), 9-19.

Schatz, M. T. (2007). High-throughput sequence alignment using Graphics Processing Units. BMC Bioinformatics, 8, art. no. 474.

Schiwietz, T. C.-C. (2006). MR image reconstruction using the GPU. Progress in Biomedical Optics and Imaging - Proceedings of SPIE, 6142 III, art. no. 61423.

Scott Le Granda, A. W. (2013). SPFP: Speed without compromise—A mixed precision model for GPU accelerated molecular dynamics simulations. Computer Physics Communications Volume 184, Issue 2, 374–380.

Seider, W. S. (2014). Design for process safety - A perspective. Computer Aided Chemical Engineering, 34, 795-800.

Sengupta, S. H. (2007). Scan primitives for GPU computing. Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware, (pp. 97-106).

Setoain, J. P. (2008). GPU for parallel on-board hyperspectral image processing. International Journal of High Performance Computing Applications, 22 (4), 424-437.

Shams, R. S. (2010). A survey of medical image registration on multicore and the GPU. IEEE Signal Processing Magazine, 27 (2), art. no. 5438962, 50-60.

Sharma, S. O.-B.-M. (2012). Evaluation of an integrated knowledge discovery and data mining process model. Expert Systems with Applications, 39 (13), 11335-11348.

Sharp, G. K. (2007). GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration. Physics in Medicine and Biology, 52 (19), art. no. 003, 5771-5783.

Sharp, T. (2008). Implementing decision trees and forests on a GPU. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 5305 LNCS (PART 4), (pp. 595-608).

Shi, Y. G.-W. (2011). Redesigning combustion modeling algorithms for the Graphics Processing Unit (GPU): Chemical kinetic rate evaluation and ordinary differential equation integration. Combustion and Flame, 158 (5), 836-847.

Shinn, A. V. (2010). Direct numerical simulation of turbulent flow in a square duct using a Graphics Processing Unit (GPU). 40th AIAA Fluid Dynamics Conference, art. no. 2010-5029.

Shiue, L.-J. J. (2005). A realtime GPU subdivision kernel. ACM Transactions on Graphics, 24 (3), (pp. 1010-1015).

Sidiropoulos, K. G. (2012). Real time decision support system for diagnosis of rare cancers, trained in parallel, on a graphics processing unit. Computers in Biology and Medicine, 42 (4), 376-386.

Silva, J. F. (2013). Data stream clustering: A survey. ACM Computing Surveys, 46 (1), art. no. 13.

Simon, L. A. (2010). Bulk video imaging based multivariate image analysis, process control chart and acoustic signal assisted nucleation detection. Chemical Engineering Science, 65 (17), 4983-4995.

Singh, A. T. (2014). Classical and neural network-based approach of model predictive control for binary continuous distillation column. Chemical Product and Process Modeling, 9 (1), 71-87.

Song, M. W. (2005). Highly efficient incremental estimation of Gaussian mixture models for online data stream clustering. Proceedings of SPIE - The International Society for Optical Engineering, 5803, art. no. 24, (pp. 174-183).

Srinivasan, R. W. (2004). Dynamic Principal Component Analysis Based Methodology for Clustering Process States in Agile Chemical Plants. Industrial and Engineering Chemistry Research, 2123-2139.

Statz, C. M. (2013). Hybrid CPU-GPU computation of adjoint derivatives in time domain. CEM 2013 - Computational Electromagnetics International Workshop, art. no. 6617123, (pp. 32-33).

Stone, J. H. (2010). GPU-accelerated molecular modeling coming of age. Journal of Molecular Graphics and Modelling, 29 (2), 116-125.

Strigl, D. K. (2010). Performance and scalability of GPU-based convolutional neural networks. Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2010, art. no. 5452452, (pp. 317-324).

Swisher, J. R. (2000). Survey of simulation optimization techniques and procedures. Winter Simulation Conference Proceedings, 1, (pp. 119-128).

T. Kourti, J. M. (1995). Process analysis, monitoring and diagnosis, using multivariate projection methods. Chemom. Intell. Lab. Syst. 28, 3-21.

Takama, N. K. (1980). Optimal water allocation in a petroleum refinery. Computers and Chemical Engineering, 251-258.

Takizawa, H. K. (2006). Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing. Journal of Supercomputing, 36 (3), (pp. 219-234).

Tan, J. A. (2010). Simulation-optimization for business decision support in a global specialty chemicals enterprise. Computer Aided Chemical Engineering, 28 (C), 133-138.

Tarabalka, Y. H. (2009). Real-time anomaly detection in hyperspectral images using multivariate normal mixture models and GPU processing. Journal of Real-Time Image Processing, 4 (3), 287-300.

Tarditi, D. P. (2006). Accelerator: Using data parallelism to program GPUs for general-purpose uses. International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS, (pp. 325-335).

Taylor, Z. C. (2008). High-speed nonlinear finite element analysis for surgical simulation using graphics processing units. IEEE Transactions on Medical Imaging, 27 (5), art. no. 4388142, 650-663.

Thrust. (n.d.). Thrust: C++ Template Library for CUDA. Retrieved 5 12, 2014, from https://code.google.com/p/thrust/

Tosukhowong, T. L. (2004). An introduction to a dynamic plant-wide optimization strategy for an integrated plant. Computers and Chemical Engineering, 199-208.

Tousain, R. B. (2006). Market-oriented scheduling and economic optimization of continuous multi-grade chemical processes. Journal of Process Control, 16 (3), (pp. 291-302).

Tsui, S.-R. W.-J.-S.-T. (2012). Parallel clustering based on partitions of local minimal-spanning-trees. Proceedings - International Symposium on Parallel Architectures, Algorithms and Programming, PAAP, art. no. 6424745, (pp. 111-118).

Tu, Y.-C. K. (2013). Data management systems on GPUs: Promises and challenges. ACM International Conference Proceeding Series, art. no. 33.

U. Fayyad, G. P.-S. (1999). Knowledge Discovery and Data Mining: Towards a Unifying Framework. Proceedings of the Second International Conference on Knowledge Discovery. Portland: AAAI Press.

U. Fayyad, G. P.-S. (November 1996). The KDD process of extracting useful knowledge from volumes of data. Communications of the ACM, 39(11), 27–34.

Vogel, D. J. (1993). A plant-wide industrial process control problem. Computers & Chemical Engineering, volume 17, No. 3, 245-255.

Wan, R. G. (2012). Weighted fuzzy-possibilistic c-means over large data sets. International Journal of Data Warehousing and Mining, 8 (4), 82-107.

Wang, J. G. (2013). Integrated intelligent control method of coke oven collector pressure. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7952 LNCS (PART 2), (pp. 246-252).

Wang, L.-F. S.-Y. (2013). Simulation optimization: a review on theory and applications. Zidonghua Xuebao/Acta Automatica Sinica, 39 (11), 1957-1968.

Wang, X. M. (1998). Automatic classification for mining process operational data. Industrial and Engineering Chemistry Research, 2215-2222.

Wasif, M. N. (2011). Scalable clustering using multiple GPUs. 18th International Conference on High Performance Computing, HiPC 2011, art. no. 6152713.

Wojek, C. D. (2008). Sliding-windows for rapid object class localization: A parallel technique. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 5096 LNCS, (pp. 71-81).

Wong, H. P.-M.-A. (2010). Demystifying GPU microarchitecture through microbenchmarking. ISPASS 2010 - IEEE International Symposium on Performance Analysis of Systems and Software, art. no. 5452013, (pp. 235-246).

Wong, T.-T. L.-S.-A. (2007). Discrete wavelet transform on consumer-level graphics hardware. IEEE Transactions on Multimedia, 9 (3), 668-673.

Woo, Y. Y. (2013). Fast PCA-based face recognition on GPUs. ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings, art. no. 6638138, (pp. 2659-2663).

Wu, R. Z. (2009). Clustering billions of data points using GPUs. Proc. Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop, UCHPC-MAW '09, Co-located with the 2009 ACM Int. Conf. on Computing Frontiers, CF'09, (pp. 1-5).

Xiong, H. P. (2006). Enhancing data analysis with noise removal. IEEE Transactions on Knowledge and Data Engineering, 18 (3), 304-319.

Xiong, Q. L. (2010). Direct numerical simulation of sub-grid structures in gas-solid flow-GPU implementation of macro-scale pseudo-particle modeling. Chemical Engineering Science, 65 (19), 5356-5365.

Xu, M. C. (2012). Discrete particle simulation of gas-solid two-phase flows with multi-scale CPU-GPU hybrid computation. Chemical Engineering Journal, 207-208, 746-757.

Xu, R. W. (2005). Survey of clustering algorithms. IEEE Transactions on Neural Networks, 16 (3), 645-678.

Xu, Z. Z. (2012). Research on clustering algorithm for massive data based on hadoop platform. Proceedings - 2012 International Conference on Computer Science and Service System, CSSS 2012, art. no. 6394257, (pp. 43-45).

Yang, Y.-F. W.-M. (2012). Parallel hierarchical K-means clustering-based image index construction method. Proceedings - 11th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, DCABES 2012, art. no. 6385323, (pp. 424-428).

Yee, T. G. (1990). Simultaneous optimization models for heat integration-II. Heat exchanger network synthesis. Computers and Chemical Engineering, 1165-1184.

Yip, W. M. (2004). The effect of model fidelity on real-time optimization performance. Computers and Chemical Engineering (pp. 267-280). Elsevier BV.

Yogita, T. D. (2013). Clustering techniques for streaming data-a survey. Proceedings of the 2013 3rd IEEE International Advance Computing Conference, IACC 2013, art. no. 6514355, (pp. 951-956).

Yoon, S. M. (2000). Statistical and causal model-based approaches to fault detection and isolation. AIChE Journal, 1813-1824.

You, Y. F. (2014). Scaling Support Vector Machines on modern HPC platforms. Journal of Parallel and Distributed Computing, . Article in Press. .

Yu, H. M. (2004). Monitoring flames in an industrial boiler using multivariate image analysis. AIChE Journal, 50 (7), 1474-1483.

Yu, J. (2011). Localized Fisher discriminant analysis based complex chemical process monitoring. AIChE Journal, 1817-1828.

Yu, J. Q. (2008). Multimode process monitoring with bayesian inference-based finite Gaussian mixture models. AIChE Journal, 1811-1829.

Yue, H. T. (2004). Weighted principal component analysis and its applications to improve FDC performance . IEEE Conference on Decision and Control (pp. 4262-4267). Nassau: Institute of Electrical and Electronics Engineers Inc.

Zaspel, P. G. (2013). Solving incompressible two-phase flows on multi-GPU clusters. Computers and Fluids, 80 (1), 356-364.

Zhang, J. L. (2012). Implmentation of a covariance-based principal component analysis algorithm for hyperspectral imaging applications with multi-threading in both CPU and GPU. International Geoscience and Remote Sensing Symposium (IGARSS), art. no. 6351726, (pp. 4264-4266).

Zhang, K. L. (2011). GPU accelerate parallel Odd-Even merge sort: An OpenCL method. Proceedings of the 2011 15th International Conference on Computer Supported Cooperative Work in Design, CSCWD , (pp. art. no. 5960058, pp. 76-83).

Zhang, T. R. (1996). BIRCH: An Efficient Data Clustering Method for Very Large Databases. SIGMOD Record (ACM Special Interest Group on Management of Data), 25 (2), 103-114.

Zhang, Y. (2009). Enhanced statistical analysis of nonlinear processes using KPCA, KICA and SVM. Chemical Engineering Science, 64 (5), 801-811.

Zhang, Y. J. (2008). Improved nonlinear fault detection technique and statistical analysis. AIChE Journal, 54 (12), 3207-3220.

Zhang, Y. M. (2002). Real-time optimization under parametric uncertainty: A probability constrained approach. Journal of Process Control, 373-389.

Zhang, Y. S. (2005). Association rules mining based on SVM and its application in simulated moving bed PX adsorption process. Chinese Journal of Chemical Engineering, 751-757.

Zhiyi, Y. Y. (2008). Parallel image processing based on CUDA. Proceedings - International Conference on Computer Science and Software Engineering, CSSE 2008, 3, art. no. 4722322, (pp. 198-201).

Zhou, Y. T. (2009). GPU-based parallel particle swarm optimization. 2009 IEEE Congress on Evolutionary Computation, CEC 2009, art. no. 4983119, (pp. 1493-1500).

Zhu, W. C. (2009). Parallel ant colony for nonlinear function optimization with graphics hardware acceleration. Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics, art. no. 5346870, (pp. 1803-1808).

Zhu, W. C. (2010). SIMD tabu search for the quadratic assignment problem with graphics hardware acceleration. International Journal of Production Research, 48 (4), 1035-1047.

Žliobaite, I. G. (2014). Adaptive preprocessing for streaming data. IEEE Transactions on Knowledge and Data Engineering, 26 (2), art. no. 6247432, 309-321.

Publications based on this work

Journal Publication

- **Lau Mai Chan**, Rajagopalan Srinivasan. A Hybrid CPU-Graphics Processing Unit (GPU) Approach for Computationally Efficient Simulation-Optimization. Manuscript submitted for publication.

Conference Publications - Paper

- **Lau Mai Chan**, Rajagopalan Srinivasan. A Graphic Processing Unit (GPU) Algorithm for Improved Variable Selection in Multivariate Process Monitoring. Computer Aided Chemical Engineering, 31, 2012.

Conference Publications - Oral

- **Lau, M. C.** & Srinivasan, R. A Graphic Processing Unit (GPU) Algorithm for Improved Variable Selection in Multivariate Process Monitoring. PSE 2012, Singapore.
- **Lau, M. C.** & Srinivasan, R., Real-time Particle Size Estimation for Crystallization Processes through GPU-based Multivariate Image Analysis. AIChE Annual Meeting 2013, San Francisco, CA.

Conference Publications - Poster

- **Lau, M. C.** & Srinivasan, R. A Graphic Processing Unit (GPU) Algorithm for Improved Variable Selection in Multivariate Process Monitoring. AIChE Annual Meeting 2012, Pittsburgh, PA.